

## **URI (Uniform Resource Identifier)**

Sottoclassi di URI:

Classificazione URI:

Operazioni sulle URI gerarchiche:

## **Protocollo HTTP**

HTTP v1.0

HTTP v1.1

HTTP v1.1 e pipelining

HTTP v2.0

Messaggi

Comandi della richiesta

GET

POST

PUT

DELETE

HEAD

OPTIONS

TRACE

Codici di stato

I cookie

Struttura dei cookie

Autenticazione

Sicurezza

Proprietà:

Architetture web

Proxy

Gateway

Tunnel

User Agent Cache

HTTP e cache

## **Linguaggio HTML**

SGML (Standard Generalized Markup Language)

Tipi MIME (Multipurpose Internet Mail Extension)

Struttura base di un documento HTML

Header

Elementi `<meta>`

Body

Contenitori di testo

Horizontal rule

Stili del testo

Tag fisici

Tag logici

Font

Liste

Liste non ordinate

Liste ordinate

Liste di definizione

Tabelle

Righe

Testate e celle

Link ipertestuali

Immagini

Form

Input

Opzioni

Textarea

- Fieldset
- Label
- Iframe
- HTML5
  - Audio e video
- CSS (Cascading Style Sheets)**
  - Selettore
  - Proprietà
  - Valori
  - Regole di risoluzione dei conflitti
- CSS 3
- XML (eXtensible Markup Language)**
  - Documento ben formato
  - Documento valido
  - Tipi semplici
    - Facets
      - Intervalli
      - Lunghezza delle stringhe
      - Rappresentazione di vincoli numerici
      - Enumerazione
      - Pattern
      - Whitespace
  - Tipi complessi
    - Elementi figlio
    - Elementi attributo
  - XSL (eXtensible Stylesheet Language)
  - XSLT (XSL for Transformation)
  - XML e Java
    - Modelli di parsing XML
  - JAXP: factory e parser
    - Parser SAX
      - Interfacce Parser-to-Application (callback)
      - Interfacce Application-to-Parser
      - Interfacce ausiliarie
    - Parser DOM
- Modelli web**
  - Modello statico
  - Modello dinamico
    - CGI (Common Gateway Interface)
    - Stato
    - Architettura frequente nei sistemi web
    - Distribuzione verticale e orizzontale
      - Replicazione: Web server
      - Replicazione: applicazione
      - Replicazione: database
    - Struttura multilivello
  - Architettura Java J2EE
    - Servlet
      - Deployment
      - Servlet context
      - Inclusione
      - Forward
    - JSP (Java Server Pages)
      - Tag
      - Built-in objects
      - Azioni
      - Custom tag
- Javascript

Inserimento codice javascript in una pagina HTML

Costruzione dinamica della pagina

Modello ad eventi e interattività

Ajax

XMLHttpRequest

Metodi di invio della richiesta

Proprietà

Metodi per la ricezione della risposta

JSON (JavaScript Object Notation)

JSON e AJAX

Accesso diretto alle basi di dati

ODBC

JDBC

Interfaccia Driver

Classe DriverManager

Interfaccia Connection

Interfaccia Statement

Interfaccia PreparedStatement

Interfaccia CallableStatement

Interfaccia ResultSet

JDBC e DB2

Conflitto di impedenza

Forza bruta

Pattern DAO (Data Access Object)

Modello retrieve : ricostruzione degli oggetti dal DB

Lazy load

Caricamento Eager

Modello create: rendere persistenti gli oggetti

ORM (Object Relational Mapping)

SessionFactory

Session

Transaction

ConnectionProvider

Transactionfactory

Oggetti persistent

Oggetti transient e detached

Ciclo di vita degli oggetti persistenti

Strategie di fetching

Strategie di caching

Hibernate in azione

Transazioni

Controllo di concorrenza

Livelli di isolamento

Transazioni in JDBC

Java Transaction API (JTA)

Transazioni in Hibernate

Java Model 2, J2EE, EJB

EJB

Session Bean

Entity Bean

Message-Driven Bean (MDB)

Servizi container-based

Spring

Inversion of Control (IoC) o Dependency injection

Persistenza

Integrazione con Web tier

Supporto a Aspect Oriented Programming

Supporto a MVC in Spring

# URI (Uniform Resource Identifier)

Gli URI forniscono un meccanismo semplice ed estensibile per identificare una risorsa. Come da nome gli identificatori sono uniformi, cioè hanno una sintassi standard:

```
<scheme>:<scheme-specific-part>
```

per la componente specifica non esiste una semantica comune a tutti gli URI ma un sottoinsieme che condivide una sintassi comune per rappresentare relazioni gerarchiche in uno spazio di nomi.

```
<scheme>://<authority><path>?<query>
```

## Sottoclassi di URI:

- Uniform Resource Name (URN): Rimangono validi anche nel caso in cui la risorsa non sia più disponibile (quindi come concetto di risorsa) un esempio è il codice ISBN dei libri. Un altro esempio è il codice fiscale. Quindi non ci dà l'informazione su dove si trova o se è ancora presente la risorsa ma ce la identifica.
- Uniform Resource Locator (URL): Parte del nome dà già un'informazione di qual è la locazione della risorsa (meccanismo di accesso primario alla risorsa). Esempio di URL:

```
<protocol>://[<username>:<password>@]<host>[:<port>][/[?][#fragment]]
```

Questa forma vale per diversi protocolli (tag `<protocol>`) di uso comune: HTTP, HTTPS, FTP, WAP ma non ad esempio per la posta elettronica.

## Classificazione URI:

- URI opaca : non soggetta a ulteriori operazioni di parsing (es. `mailto:paolo.rossi@disi.unibo.it`)
- URI gerarchica : soggetta a ulteriori operazioni di parsing per esempio per separare l'indirizzo del server dal percorso all'interno file system.

## Operazioni sulle URI gerarchiche:

- Normalizzazione: Processo di rimozione dei segmenti ".", ".." e altri caratteri speciali dal path
- Risoluzione : la URI originaria viene risolta basandosi su una terza URI, detta base URI.

(es. URI originaria: `docs/doc.html#28` , Base URI: <http://disi.unibo.it/> , Risultato: <http://disi.unibo.it/docs/doc.html#28>)

- Relativizzazione : è il processo inverso alla risoluzione

---

# Protocollo HTTP

## HTTP v1.0

- request-response (unica interazione possibile: il cliente fa la richiesta e il server risponde)
- stateless (non c'è mantenimento dello stato)
- one-shot (La connessione viene utilizzata solo per la richiesta e per la risposta e poi viene chiusa)

Estremamente semplice e veloce da sviluppare ma assolutamente inefficiente anche per documenti molto semplici. Consideriamo ad esempio una pagina contenente un documento HTML con 10 immagini. Viene dunque creata la connessione TCP, viene effettuata la richiesta del documento HTML con i relativi riferimenti alle immagini ma una volta ricevuto viene chiusa la connessione e ripetuto il ciclo con relativa riapertura della connessione per ogni immagine da inviare.

## HTTP v1.1

Il cliente può richiedere al servitore di lasciare aperta la connessione per interazioni successive.

### HTTP v1.1 e pipelining

Per migliorare ulteriormente le prestazioni si può usare la tecnica del pipelining che consiste nell'invio di molteplici richieste da parte del client prima di terminare la ricezione delle risposte. Le risposte debbono però essere in ordine con le richieste.

## HTTP v2.0

- request-response multiplexing (il cliente/servitore può mandare pacchetti mentre il servitore/cliente ne sta già inviando altri)
- header compression
- server push (il server è in grado di comunicare con il cliente senza necessariamente aver ricevuto una precedente richiesta)

HTTP/2 ora è un protocollo standard ma nasce come protocollo proprietario ideato da Google con nome SPDY.

## Messaggi

Un messaggio HTTP è definito da due strutture:

- Header: contiene tutte le informazioni necessarie per l'identificazione del messaggio
- Body: contiene i dati trasportati dal messaggio

I dati sono codificati secondo il formato specificato nell'header (solitamente MIME)

Esempio di header:

```
GET /somedir/page.html HTTP/1.1
Host: www.unibo.it
Connection: close
User-agent: Chrome/37.0
Accept: text/html, image/gif, image/jpeg
Accept-language: fr
```

## Comandi della richiesta

### GET

Serve per richiedere una risorsa al server. E' il metodo più frequente e prevede il passaggio di parametri (la parte dell'URL).

Poiché l'URI ha una lunghezza limitata, il numero di parametri che è possibile passare è ristretto.

### POST

Serve per richiedere una risorsa al server ma a differenza del GET, i dettagli per l'identificazione ed elaborazione della risorsa non sono nell'URL ma sono contenuti nel body del messaggio. Non ci sono quindi limiti di lunghezza nei parametri di una richiesta.

### PUT

Chiede la memorizzazione sul server di una risorsa all'URL specificato. Normalmente disabilitato sui server pubblici

## DELETE

Richiede la cancellazione della risorsa riferita dall'URL specificato. Normalmente disabilitato sui server pubblici

## HEAD

E' simile a GET ma il server deve rispondere soltanto con gli header relativi, senza body (utilizzato per verificare un URL)

## OPTIONS

Serve per richiedere informazioni sulle opzioni disponibili per la comunicazione

## TRACE

Usato per invocare il loop-back remoto a livello applicativo del messaggio di richiesta

## Codici di stato

Il codice di stato è un numero a tre cifre, di cui la prima indica la classe della risposta:

- 1xx : Informational. Una risposta temporanea alla richiesta, durante il suo svolgimento
- 2xx : Successfull. Il server ha ricevuto, capito e accettato la richiesta
- 3xx : Redirection. Il server ha ricevuto e capito la richiesta ma sono necessarie altre azioni del client per portare a termine la richiesta
- 4xx : Client error. La richiesta del client non può essere soddisfatta per un errore da parte del client (errore sintattico o richiesta non autorizzata)
- 5xx : Server error: La richiesta può anche essere corretta, ma il server non è in grado di soddisfare la richiesta per un problema interno

## I cookie

Il protocollo prevede una struttura dati che si muove come un token, dal client al server e viceversa: i cookie.

Hanno come scopo quello di fornire un supporto per il mantenimento dello stato dato che HTTP è stateless.

Possono essere generati dal client o dal server e sono semplicemente delle stringhe di max 255 caratteri.

Poiché essi vengono salvati sul browser, cambiando macchina o browser i cookie vanno persi.

## Struttura dei cookie

- Key : identifica univocamente un cookie all'interno di un path
- Value : valore associato al cookie
- Path : posizione nell'albero di un sito al quale è associato
- Domain: dominio dove è stato generato
- Max-age (opzionale) : numero di secondi di vita
- Secure (opzionale) : trasferiti solo se il protocollo è sicuro
- Version: identifica la versione del protocollo di gestione dei cookie

## Autenticazione

Nel caso in cui un cliente tenti di accedere ad una risorsa sul server che richiede un'autenticazione, quest'ultimo genera una risposta (Challenge) con codice di errore 401 e può creare nel browser una form dove l'utente può inserire le credenziali. Normalmente per l'invio delle credenziali si usa il metodo POST per evitare che siano trasmesse "in chiaro".

## Sicurezza

Sicurezza del canale di trasporto:

- SSL : Secure Socket Layer

- TLS : Transport Layer Security (Sostituisce SSL ed è alla base di HTTPS)

### Proprietà:

- Confidenzialità : un attaccante non è in grado di interpretare il messaggio, ma solo il diretto interessato
- Integrità : nel caso il messaggio venga modificato in qualche modo dall'attaccante, l'errore viene individuato
- Autenticità : ciò che viaggia sul canale può essere prodotto solo dai partecipanti al canale

## Architetture web

### Proxy

Applicativo in grado di agire sia come client che come server al fine di effettuare richieste per conto di altri clienti.

Ci permette di definire delle white/black-list o fare cache (**Proxy Cache**) come un filtro per il nostro terminale.

### Gateway

Si occupa di ricevere le richieste dei clienti e smistarle tra i vari server.

Se ci sono richieste popolari può fare lui stesso da cache evitando di richiedere più volte la stessa cosa ai server (**Reverse Proxy Cache**).

### Tunnel

Programma applicativo che agisce come “blind relay” tra due connessioni.

### User Agent Cache

Lo user agent (tipicamente il browser) mantiene una cache delle pagine visitate dall'utente

## HTTP e cache

In HTTP non c'è nessun protocollo per la gestione della cache ma si possono sfruttare dei meccanismi esistenti che possono aiutare:

- Freshness : Attraverso degli attributi quali Expires response header lato server e Cache Control : max-age lato client possiamo pensare che la pagina in cache diventi obsoleta dopo quel dato intervallo di tempo
- Validation : il browser potrebbe inviare a intervalli di tempo delle richieste HEAD per ottenere l'header di risposta dal server e controllare se sia una versione più
- Invalidation : facendo operazioni di POST/PUT/DELETE la pagina in cache viene invalidata

---

# Linguaggio HTML

HTML, linguaggio ideato da Tim Berners-Lee a inizio anni '90, definisce documenti di testo quindi necessita di sistemi di codifica caratteri:

- ASCII
- Famiglia ISO 8859/ANSI (8 bit)
- Unicode (8, 16 o 32 bit: UTF-8, UTF-16 e UTF-32)

Ma è anche un linguaggio a marcatori composto da:

- un'insieme di istruzioni dette **tag** o mark-up
- una grammatica che regola l'uso del mark-up
- una semantica che definisce il dominio di applicazione e la funzione del mark-up

I linguaggi di mark-up sono stati divisi in:

- linguaggi **procedurali** o imperativi

Il mark-up specifica quali operazioni un dato programma deve compiere su un documento per ottenere una determinata formattazione (LaTeX)

- linguaggi **dichiarativi** o descrittivi

il mark-up descrive la struttura di un documento testuale identificandone i componenti (SGML, HTML, XML)

## SGML (Standard Generalized Markup Language)

Il primo standard di linguaggio a marcatori ideato nel 1986 da Charles Goldfarb. E' un meccanismo flessibile che permette di descrivere il contenuto del documento ma non contiene sequenze di istruzioni per la formattazione.

HTML è un'applicazione SGML, ovvero un linguaggio per la rappresentazione di un tipo di documento SGML che associa anche significati grafici agli elementi che definisce.

## Tipi MIME (Multipurpose Internet Mail Extension)

Rappresenta il tipo di contenuto di un messaggio ad esempio HTML request. Un tipo MIME è spesso con questa sintassi: /tipo/sottotipo .

## Struttura base di un documento HTML

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
  <head>
    <title>Hello document</title>
  </head>
  <body>
    Hello World!
  </body>
</html>
```

Il primo elemento del documento è la Document Type Definition (DTD). Serve al browser per identificare le regole di interpretazione da applicare.

## Header

Il tag header può contenere:

- <title> titolo della pagina
- <meta> metadati informazioni utili ad applicazioni esterne
- <base> definisce come vengono gestiti i riferimenti relativi ai link
- <link> collegamenti verso file esterni: CSS, script, icone visualizzabili nella barra degli indirizzi del browser
- <script> codice eseguibile utilizzato dal documento
- <style> informazioni di stile

## Elementi <meta>

Gli elementi di tipo <meta> possono essere di due tipi, distinguibili dal primo attributo:

- **http-equiv** danno informazioni al browser su come gestire la pagina

```
<meta http-equiv=nome content=valore>
```

Alcuni esempi di nomi sono:

- refresh
- expires



- content type
- **name** forniscono informazioni utili ma non critiche

```
<meta name=nome content=valore>
```

Alcuni esempi di nomi sono:

- author
- description
- copyright
- keywords
- date

## Body

I tipi di elementi del body sono

- Intestazioni
- Strutture di testo
- Aspetto del testo
- Elenchi e liste
- Tabelle
- Form
- Collegamenti ipertestuali e ancore
- Immagini e contenuti multimediali
- Contenuti interattivi

Dal punto di vista del layout della pagina gli elementi HTML si dividono in:

- **Elementi "block level"** : costituiscono un blocco attorno a sé e di conseguenza vanno a capo (paragrafi, tabelle, form, ...)
- **Elementi "inline"** : non vanno a capo e possono essere integrati nel testo (link, immagini, ...)
- **Liste** : numerate o puntate

Un elemento block-level può contenere altri elementi dello stesso tipo o di tipo inline.

Un tipo inline può contenere solo altri elementi inline

Un'altra distinzione è quella tra elementi **rimpiazzati**, di cui il browser conosce le dimensioni intrinseche, ed elementi non rimpiazzati. Alcuni esempi di elementi rimpiazzati sono `<img>` , `<input>` , `<textarea>` , `<select>`

## Contenitori di testo

Sono contenitori di testo:

- `<p>` elemento block-level che lascia spazio prima e dopo la propria chiusura
- `<div>` elemento block-level che non lascia spazio prima e dopo la chiusura (elemento di tipo blocco per eccellenza)
- `<span>` elemento inline e quindi non va a capo. E' utilizzato all'interno di div insieme ai fogli di stile per dare un aspetto particolare ad un pezzo di testo come ad esempio evidenziare.

## Horizontal rule

Serve per definire una riga di separazione

```
<hr align="{left|center|right}" size="n" width="n" noshade>
```

## Stili del testo

### Tag fisici

<code>&lt;tt&gt;</code>	Monospaziato
<code>&lt;b&gt;</code>	Grassetto
<code>&lt;i&gt;</code>	Corsivo
<code>&lt;u&gt;</code>	Sottolineato (deprecato)
<code>&lt;s&gt;</code>	Barrato (deprecato)

## Tag logici

<code>&lt;strong&gt;</code>	Grassetto
<code>&lt;em&gt;</code>	Corsivo
<code>&lt;code&gt;</code>	Codice (monospaziato)
<code>&lt;kbd&gt;</code>	Keyboard (monospaziato)
<code>&lt;abbr&gt;</code>	Abbreviazione
<code>&lt;acronym&gt;</code>	Acronimo
<code>&lt;address&gt;</code>	Indirizzo fisico o e-mail (corsivo)
<code>&lt;blockquote&gt;</code>	Citazione (rientrato)
<code>&lt;cite&gt;</code>	Citazione (corsivo)

## Font

```
<font size="[+|-]n" color="color" face="font"></font>
```

## Liste

### Liste non ordinate

```
<ul type="{disc|circle|square}">
  <li></li>
  <li></li>
  ...
</ul>
```

### Liste ordinate

```
<ol type="{1|a|A|i|I}">
  <li></li>
  <li></li>
  ...
</ol>
```

### Liste di definizione

```
<dl>
  <dt>TERMINE1</dt>
  <dd>Definizione di Termine1</dd>
  <dt>TERMINE2</dt>
  <dd>Definizione di Termine2</dd>
  ...
</dl>
```

## Tabelle

```
<table align="{left|center|right}" width="{n|n%}" bgcolor="#xxxxxx" border="n"
cellspacing="n" cellpadding="n"></table>
```

### Righe

```
<tr align="{left|center|right|justify}" valign="{top|middle|bottom|baseline}"
bgcolor="#xxxxxx"></tr>
```

### Testate e celle

```

<!-- Definire le celle della testata-->
<th align="{left|center|right|justify}" valign="{top|middle|bottom|baseline}"
bgcolor="#xxxxxx" width="{n|n%}" height="{n|n%}" rowspan="n" colspan="n"></th>

<!-- Definire le celle del contenuto-->
<td align="{left|center|right|justify}" valign="{top|middle|bottom|baseline}"
bgcolor="#xxxxxx" width="{n|n%}" height="{n|n%}" rowspan="n" colspan="n"></td>

```

## Link ipertestuali

*Un link in HTML è definito attraverso delle ancore*

Il tag per definire un' ancora è `<a>`. L' ancora di origine è caratterizzata da un attributo `href` mentre quella di destinazione da un attributo `name`:

```

<a href="file.html#section1"></a>

<a name="section1"></a>

```

## Immagini

```



```

## Form

```

<form action="uri" name="text" method="{get|post}" enctype="content-type"></form>

```

Se non è specificato allora `enctype = "application/x-www-form-urlencoded"`

## Input

- text

```

<input type="text" name="text" size="n" maxlen="n">

```

- password

```

<input type="password" name="text" size="n" maxlen="n">

```

- file

```

<!-- E' necessario specificare un enctype di tipo multipart per definire che il
nostro file HTML contiene sia elementi di testo che elementi binari-->
<form enctype="multipart/form-data">
    <input type="file" name="text" value="content-type">
</form>

```

- checkbox

```

<input type="checkbox" name="text" value="text-val" checked="checked">

```

- radio

```

<input type="radio" name="text" value="text-val" checked="checked">

```

- submit

```
<input type="submit" value=submit>
```

- reset

```
<input type="reset" value=reset>
```

- button

```
<button type="button"> text </button>
```

- hidden

```
<input type="hidden" value=hidden>
```

## Opzioni

```
<select name="text" multiple="multiple">
  <!-- il tag <optgroup> è opzionale e serve nelle liste lunghe per dividerle in
  gruppi -->
  <optgroup label="text-val">
    <option value="text-val" selected="selected"> val1 </option>
  </optgroup>
  ...
</select>
```

## Textarea

```
<textarea name="text" rows="n" cols="n"> Testo </textarea>
```

## Fieldset

```
<fieldset>
  <legend> Legenda </legend>
  ...
</fieldset>
```

## Label

```
<label> Etichetta </label>
```

## Iframe

```
<iframe src="file.html" width="n%">
</iframe>
```

# HTML5

## Audio e video

```
<video id="id" width="n" height="n" preload controls="{metadata|none|auto}">
  <source src="movie.mp4" type="video/mp4" codecs="avc1.42E01E, mp4a.40.2" >
  ...
</video>

<audio id="id" width="n" height="n" preload controls="{metadata|none|auto}">
  <source src="movie.mp4" type="audio/mp3" codecs="..." >
  ...
</audio>
```

# CSS (Cascading Style Sheets)

I fogli di stile a cascata hanno lo scopo fondamentale di separare contenuto e presentazione nelle pagine Web. La parola chiave del linguaggio CSS è "cascading", ciò prevede la presenza di fogli di stile multipli, che agiscono uno dopo l'altro, in cascata, per indicare le caratteristiche tipografiche e di layout di un documento HTML.

Gli obiettivi principali del linguaggio CSS sono:

- Ridurre tempi di scaricamento delle pagine : una pagina che usa i CSS è meno della metà di una pagina che usa la formattazione con tag HTML, inoltre se il file CSS è condiviso da più pagine viene scaricato una volta sola
- Ripulire il codice HTML e rendere il contenuto riusabile
- Rendere le pagine visualizzabili e usabili da dispositivi non convenzionali: laptop, palmari, smartphone, ecc.

Per definire il riferimento ad un file CSS in un file HTML occorre usare il tag `<link>` all'interno dell'header:

```
<head>
  <link rel="stylesheet" href="file.css" type="text/css">

  <!-- In alternativa si può utilizzare il tag <style> ma è ormai deprecato -->
  <style type="text/css">@import url(file.css);</style>
</head>
<body>
  <!-- E' possibile anche definire degli stili specifici in linea, ovviamente
  l'operazione è altamente sconsigliata! -->
  <h1 style="color: red"> Titolo1 </h1>
</body>
```

Una regola CSS ha un'espressione del tipo:

```
H1{color: blue}
```

- Selettore: H1
- Dichiarazione: color: blue
  - Proprietà: color
  - Valore: blue

## Selettore

- selettore universale

```
*{ ... }
```

- selettore di tipo

```
tipo_elemento { ... }
```

- classi

```
.nome_classe{ ... } /* Se non si specifica il selettore davanti il browser
sottointende il selettore universale */
```

- identificatori

```
#nome_id { ... } /* Se non si specifica il selettore davanti il browser
sottointende il selettore universale */
```

- Pseudoclassi

```
tipo_elemento:proprietà { ... }

/* Ad esempio: */
a:link { ... }
h1:hover { ... }
```

- **Pseudoelementi**

```
tipo_elemento:parte { ... }

/* Ad esempio: */
p:first-line { ... }
p:first-letter { ... }
```

- **selettori gerarchici**

```
tipo1 tipo2 { ... }      /* tipo2 discende da tipo1 */
tipo1>tipo2 { ... }      /* tipo2 è figlio di tipo1 */
tipo1+tipo2 { ... }      /* tipo2 è fratello di tipo1 */
```

- **raggruppamenti**

```
tipo1, tipo2, tipo3 { ... }
```

## Proprietà

```
/* Proprietà singole */
p { margin-top: 10px;
    margin-right: 8px;
    margin-bottom: 10px;
    margin-left: 8px; }

/* proprietà shorthand */
p { margin: 10px 8px 10px 8px}
```

- **Colori e sfondi**

```
/* COLOR */
foreground-color : {colore}
background-color: {colore}
border-color : {colore}

/* sintassi shorthand*/
selettore {color: foreground-color background-color border-color; }
```

```
/* BACKGROUND - sfondo applicabile a body e tabelle */
background-color : {colore | transparent}
background-image : {url | none}
background-repeat : {repeat | repeat-x | repeat-y | no-repeat}
background-attachment : {scroll | fixed}
background-position : { x,y in % o assoluti | top | right | bottom | left}

/* sintassi shorthand*/
selettore {background: background-color backgroundimage background-repeat
background-attachment background-position;}
```

---

- **Caratteri e testo**

```

/* FONT */
font-family : {Verdana | Helvetica | ... | serif | sans-serif | cursive |
fantasy | monospace} /* Si possono specificare anche più famiglie, il browser
procederà a cercare dalla prima all'ultima */
font-size : {xx-small | x-small | small | medium | large | x-large | xx-large
| smaller | larger | grandezza}
font-weight : {numero | normal | bold | bolder | lighter}
font-style : {normal | italic | oblique}
font-variant : {normal | small-caps}

/* sintassi shorthand*/
selettore {font: font-style font-variant font-weight font-size font-family}

```

```

/* LINE-HEIGHT */
line-height : {normal | numero | grandezza | percentuale}

/* TEXT-ALIGN */
text-align : {left | right | center | justify}

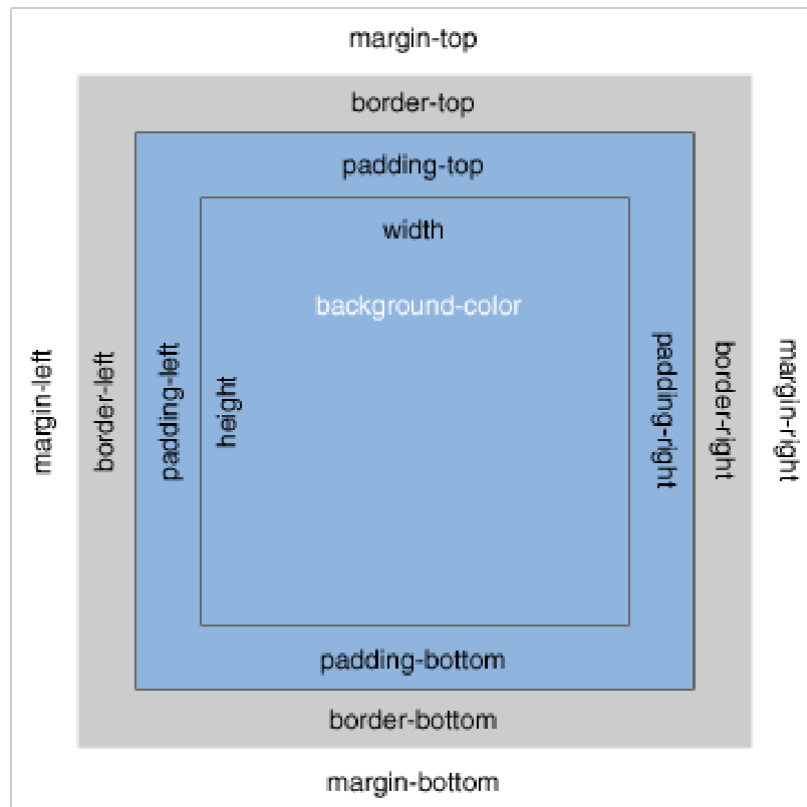
/* TEXT-DECORATION */
text-decoration : {none | underline | overline | line-through}

/* TEXT-INDENT */
text-indent : {grandezza | percentuale}

/* TEXT-TRANSFORM */
text-transform : {none | capitalize | uppercase | lowercase}

```

- Box-model



- Liste

```

/* LIST-STYLE-IMAGE - immagine per il punto dell'elenco */
list-style-image : {url | none}

/* LIST-STYLE-POSITION - posizione del punto */
list-style-position : {inside | outside}

/* LIST-STYLE-TYPE */
list-style-type : {none | disc | circle | square | decimal | decimal-leading-zero | lower-roman | upper-roman | lower-alpha | upper-alpha | lower-latin | upper-latin | lower-greek}

```

- Display e gestione degli elementi floating

```

/* DISPLAY */
display : {inline | block | list-item | none}

/* FLOAT - definire effetti di floating*/
float : {left | right | none}

/* CLEAR - cancellare gli effetti del floating*/
clear : {left | right | none | both}

```

- Posizionamento

```

/* POSITION */
position : {static | absolute | relative | fixed}

/* VISIBILITY */
visibility : {visible | hidden}

/* Z-INDEX */
z-index : {auto | numero}

```

- Tabelle

```

/* TABLE-LAYOUT */
table-layout : {auto | fixed}

/* BORDER-COLLAPSE */
border-collapse : {collapse | seprate}

/* BORDER-SPACING */
border-spacing : {grandezza}

```

---

## Valori

- *numero* : interi e reali utilizzando il punto come separatore decimale
- *grandezza*: un numero seguito da un unità di misura

Unità di misura relative:

- *em*: relativa alla dimensione del font in uso
- *px*: pixel relativi al dispositivo in uso

Unità di misura assolute

- *in*: pollici (2,54cm)
- *cm*: centimetri
- *mm*: millimetri
- *pt*: punti tipografici (1/72 di pollice)
- *pc*: pica (12 punti)
- *percentuale*: numero seguito da %








- *url*: si usa la sintassi `url(percorso)`
- *stringa*: testo delimitato da apici singoli o doppi
- *colore*:
  - forma esadecimale: `#RRGGBB`
  - funzione: `rgb(rosso, verde, blu)`

## ■ Colori assoluti:

 black - nero	 green - verde
 silver - argento	 lime - verde chiaro
 gray - grigio	 olive - oliva
 white - bianco	 yellow - giallo
 maroon - marrone	 navy - blu scuro
 red - rosso	 blue - blu
 purple - viola	 teal - verde acqua scuro
 fuchsia - fucsia	 aqua - verde acqua

## ■ Colori dipendenti dalle proprietà di sistema:

 background - il colore di sfondo del desktop
 buttonFace - il colore di sfondo dei pulsanti
 buttonText - testo dei pulsanti
 captionText - testo delle etichette
 grayText - testo disabilitato

## Regole di risoluzione dei conflitti

Il CSS assegna un peso a ciascun blocco di regole, in caso di conflitto vince quella con peso maggiore. Per determinare quella con peso maggiore si applicano una serie di regole:

- *Origine*: l'ordine di prevalenza è autore, utente, browser
- *Specificità del selettore*: ha la precedenza il selettore con specificità maggiore
- *Ordine di dichiarazione*: se esistono due dichiarazioni con ugual specificità e origine vince quella fornita per ultima

Attraverso la clausola `!important` permette ad una regola di avere precedenza sulle altre.

## CSS 3

I CSS 3 estendono le funzionalità dei CSS 2.1 mantenendone la piena compatibilità. Tra le nuove funzionalità:

- se prima potevamo selezionare un elemento in base alla presenza di un attributo: `p[class]{}` ora possiamo selezionarlo anche in base alla presenza di particolari stringhe al suo interno: `p[class^="html"]{}`
- possiamo selezionare elementi in base alla loro posizione nel DOM: `p:nth-child(3){}` o in base al loro particolare stato nella user interface: `input:disabled{}`
- supporto a XML: `@namespace element "http://www.sito.it/ns/element/"`
- pieno controllo di un layout: posizionamento, sfondi multipli, trasparenze, angoli smussati, effetti sul testo, ecc.
- consentono di indirizzare gli stili non solo ai dispositivi specifici, ma anche di specificare quali condizioni devono essere soddisfatte per l'applicazione di tali stili tramite le nuove regole: `@media screen and (max-width: 800px){}`
- creare intestazioni, piè di pagina, note in calce e numerazioni automatiche delle pagine per la stampa

# XML (eXtensible Markup Language)

XML è un formalismo (metalinguaggio) con una sintassi rigorosa e al contempo flessibile che consente la rappresentazione di documenti e dati strutturati su supporto digitale. Da XML derivano dei veri e propri linguaggi quali SVG, XHTML, MathML. Un esempio di documento XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="gree.css"?>

<prenotazione>
  <idVolo>PA321</idVolo>
  <idCliente>PP2305</idCliente>
  <data>22-10-2001</data>
  <prezzo valuta="Euro">245</prezzo>
</prenotazione>
```

I tag sono detti elementi mentre i valori dentro i tag sono detti attributi.

Un'applicazione in grado di elaborare dati in formato XML viene definita **elaboratore XML**.

Ha una rappresentazione fisica (il file con i vari tag) e una struttura logica (ad albero).

Per i file di validazione della grammatica di un file xml è possibile definirli in due modi:

```
<!--grammatica contenuta in un file locale-->
<!DOCTYPE book SYSTEM "book.dtd">
<!--grammatica accessibile ad un URL pubblico-->
<!DOCTYPE book PUBLIC "http://www.books.org/book.dtd">
```

Riferimenti ad entità:

Nome Entità	Riferimento	Carattere
lt	&lt;	<
gt	&gt;	>
amp	&amp;	&
apos	&apos;	'
quot	&quot;	"

Per inserire o porzioni di testo o codice XML/MHTML senza sostituire i caratteri speciali si possono usare le sezioni **CDATA**:

```
<![CDATA[ Contenuto della sezione ]]>
```

Il **namespace** permette attraverso un prefisso di indentificare il vocabolario di appartenenza di elementi e attributi:

```
<lb:libro xmlns:lb="mysite.com/libri">
  <au:autore xmlns:au="mysite.com/autori">
    <au:titolo>Sir</au:titolo>
    <au:nome>William Shakespeare</au:nome>
  </au:autore>
  <lb:titolo>Romeo and Juliet</lb:titolo>
</lb:libro>
```

Il namespace è valido dal punto in cui è definito in avanti a meno che non ne venga definito un altro.

## Documento ben formato

Affinché un documento XML sia ben formato:

- deve contenere una dichiarazione corretta
- il corpo deve avere un unico elemento radice
- ogni elemento deve avere un tag di apertura e uno di chiusura
- gli elementi devono essere opportunamente nidificati, cioè i tag di chiusura devono seguire l'ordine inverso dei rispettivi tag di apertura
- i nomi dei tag di apertura e chiusura devono coincidere (XML è case-sensitive)
- i valori degli attributi devono sempre essere racchiusi tra singoli o o doppi apici

## Documento valido

Uno strumento per definire grammatiche è costituito dalla Document Type Definition (DTD), un documento costituito da un elenco di dichiarazioni che descrivono la struttura del documento (elementi, contenuto degli elementi, attributi associati a ciascun elemento).

Lo strumento utilizzato durante il corso sarà **XSD** (XML Schema Definition). Un documento XSD comprende:

- Un namespace di riferimento: <http://www.w3.org/2001/XMLSchema>
- Dichiarazione di:
  - Elementi
  - Attributi
- Definizione di tipi:
  - Semplici
  - Complessi
  - Estesi

Esempio file XSD:

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- elemento root, contiene il riferimento al namespace -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  <!-- definizione di un elemento con un certo nome e tipo -->
  <xs:element name="message" type="messageType"/>
  <!-- definizione inline del tipo come una sequenza di elementi (l'ordine
  conta).
  un'altra opzione era quella di racchiudere tutto dentro il tag di
  element -->
  <xs:complexType name="messageType">
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Quindi un file XML per essere validato dal precedente file XSD deve presentarsi nella seguente forma:

```
<?xml version="1.0"?>
<message
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://mysite.it/msg.xsd"> <!-- Rif. al
file XSD precedente -->
  <to>Bob</to>
  <from>Janet</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend</body>
</message>
```

## Tipi semplici

Gli elementi di tipo semplice possono contenere solo caratteri alfanumerici e non altri elementi. La definizione di nuovi tipi avviene derivando per restrizione dai tipi predefiniti. La restrizione avviene specificando dei vincoli (*facets*):

```
<xs:simpleType name="derivedType">
  <xs:restriction base="baseType">
    {facets}
  </xs:restriction>
</xs:simpleType>
```

## Facets

*NB: in generale facets vanno in AND tra loro*

### Intervalli

```
<xs:minInclusive value="0"/>
<xs:maxInclusive value="5"/>

<xs:minExclusive value="0"/>
<xs:maxExclusive value="5"/>
```

### Lunghezza delle stringhe

```
<!-- Si applicando ai baseType xs:string (e derivati) -->
<xs:minLength value="7"/>
```

### Rappresentazione di vincoli numerici

```
<!-- Si applicando ai baseType xs:decimal (e derivati) -->
<xs:totalDigits value="2"/>
<xs:fractionDigits value="2"/> <!-- Numero massimo di cifre decimali-->
```

## Enumerazione

*Nota: enumeration va in OR con gli altri elementi enumeration (non avrebbe senso con AND)*

```
<!-- Si applicando ai baseType xs:string (e derivati) -->
<xs:enumeration value="CD"/>
<xs:enumeration value="DVD"/>
```

## Pattern

```
<xs:pattern value="[0-9]*\.[0-9]{2}"/> <!-- Costrutto più potente che fa uso delle
regex per definire tipi complessi come in questo caso la valuta in euro con
esattamente due cifre decimali -->
```

## Whitespace

Definisce come comportarsi in presenza di caratteri spaziatura:

- `preserve` Preserva i caratteri spaziatura
- `replace` Rimpiazza i caratteri spaziatura con tutti spazi
- `collapse` Rimpiazza i caratteri spaziatura con uno spazio

```
<xs:whiteSpace value="{ preserve, replace, collapse}"/>
```

## Tipi complessi

E' possibile definire:

- tipi con nome:

```
<xs:complexType name="typeName">    </xs:complexType>
```

- tipi anonimi (inline):

```
<xs:element name="myElement">    <xs:complexType>    </xs:complexType></xs:element>
```

Si può anche specificare un attributo `mixed="{true | false}"` che, se impostato, consente la presenza di caratteri oltre agli elementi all'interno di un tipo complesso.

## Elementi figlio

Gli elementi figlio possono essere:

- `sequence` : gli elementi dichiarati devono comparire nel documento istanza nell'ordine indicato e con le cardinalità specificate
- `choice` : nel documento istanza deve comparire solo uno degli elementi dichiarati
- `all` : gli elementi dichiarati devono comparire nel documento istanza almeno una volta con ordine qualsiasi

La cardinalità degli elementi figlio viene espressa mediante gli attributi

- `minOccurs` [default 1]
- `maxOccurs` [default 1]

```
<xs:complexType name="mySeq">    <xs:sequence>        <xs:element name="e1" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>        <xs:element name="e2" type="xs:string" maxOccurs="2"/>    </xs:sequence></xs:complexType>
```

## Elementi attributo

Gli elementi attributo possono essere contenuti solo da tipi complessi e possono essere:

- `name="text"`
- `type="text"`
- `use="{optional | prohibited | required}"`
- `default="value"` oppure `fixed="value"` : permettono di specificare un valore di default o un valore prefissati (sono mutuamente esclusivi)

```
<xs:complexType name="WAttrType">    <xs:attribute name="at" type="xs:string"/></xs:complexType >
```

## XSL (eXtensible Stylesheet Language)

Si occupa della trasformazione e della impaginazione di contenuti XML. Si basa principalmente su:

- XSLT (XSL for Transformations): gestisce le trasformazioni e non l'impaginazione dei contenuti
- XSL-FO (XSL Formatting Objects): orientato alla visualizzazione ed impaginazione dei contenuti (es. in PDF)
- XPath (XML Path Language): serve per costruire percorsi di ricerca di informazioni all'interno di documenti XML

## XSLT (XSL for Transformation)

XSLT è un linguaggio di programmazione a tutti gli effetti: permette di gestire variabili, parametri, cicli, condizioni, funzioni.

Lavora sulla struttura del documento:

- Costruisce l'albero del documento (DOM)
- Lo attraversa cercando le informazioni indicate
- Produce un nuovo documento (di solito XHTML) applicando le regole definite

## XML e Java

Un elemento importante che ha consentito la diffusione dei linguaggi e delle tecnologie XML è il supporto fornito da strumenti per il **parsing**. In Java: specifiche **JAXP** (Java API for XML Processing).

Compiti del parser:

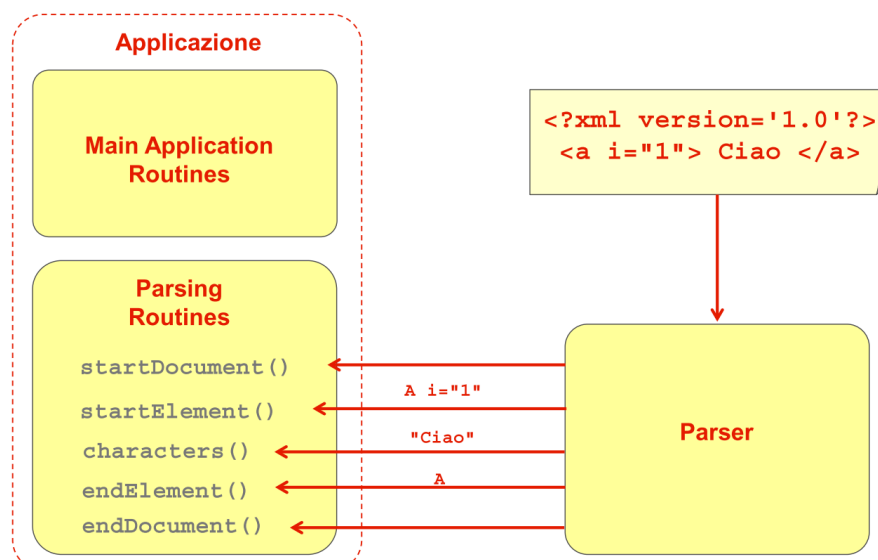
- Decomporre i documenti XML (istanza) nei loro elementi costitutivi
- Controllare che il documento sia ben formato
- Controllare eventualmente che il documento sia valido (DTD, XML Schema) [In questo caso si parla di **parser validanti**]

## Modelli di parsing XML

- Interfacce basate su eventi : Modello più efficiente basato su **Interfacce SAX** (Simple API for XML) che sfruttano un modello a *callback* ossia la lettura del documento scatena degli eventi che causano l'invocazione di metodi.

In java:

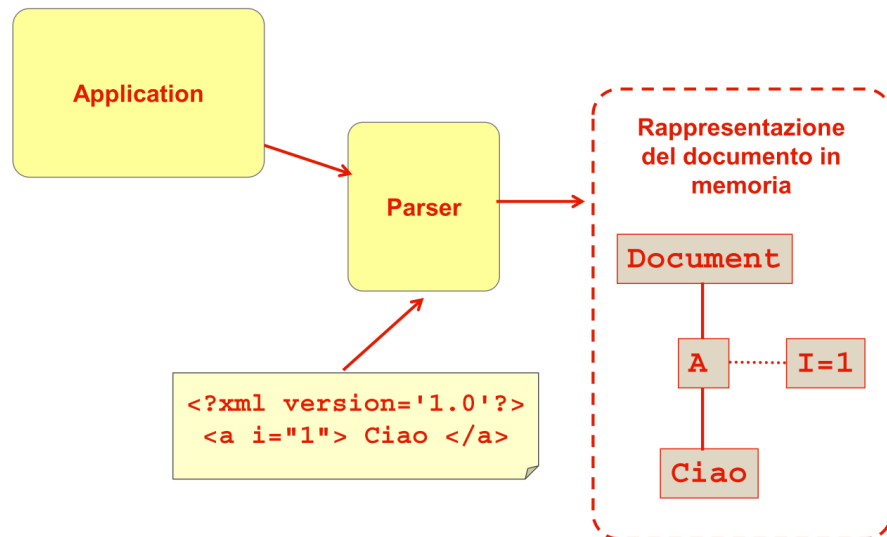
```
import org.xml.sax;
```



- Interfacce Object Model : **W3C DOM** Recommendation è un modello meno efficiente perché rappresenta l'intero documento come un albero (*parser-tree*) e lo tiene in memoria. Però è più semplice da usare per via della rappresentazione Object-Oriented.

In java:

```
import org.w3c.dom;
```



## JAXP: factory e parser

```
import javax.xml.parser;
```

### Parser SAX

#### Interfacce Parser-to-Application (callback)

- `ContentHandler`: metodi per elaborare gli eventi generati dal parser

```
setDocumentLocator(Locator locator);

startDocument();
endDocument();

startElement(String namespaceURI, String localname, String rawName,
Attributes atts);
endElement(String namespaceURI, String localname, String qName);

characters(char ch[], int start, int length)           //notifica la
presenza di character data
ignorableWhitespace(char ch[], int start, int length)  //notifica la
presenza di whitespaces ignorabili
```

- `DTDHandler`: metodi per ricevere notifiche su entità esterne al documento e loro notazione dichiarata in DTD o XML Schema
- `ErrorHandler`: metodi per gestire gli errori ed i warning nell'elaborazione di un documento
- `EntityResolver`: metodi per personalizzare l'elaborazione di riferimenti ad entità esterne

#### Interfacce Application-to-Parser

- `XMLReader`: interfaccia che consente all'applicazione di invocare il parser e di registrare gli oggetti che implementano le interfacce di callback
- `XMLFilter`: interfaccia che consente di porre in sequenza vari `XMLReaders` come una serie di filtri

#### Interfacce ausiliarie

- `Attributes`: metodi per accedere ad una lista di attributi

- `Locator`: metodi per individuare l'origine degli eventi nel parsing dei documenti (es. `systemID`, numeri di linea e di colonna, ecc.)

## Parser DOM

Interfacce:

- `Node`

```
getNodeName()    // Se applicato ad Element equivale a getTagName(), se
                  applicato ad Attr restituisce il nome dell'attributo
getNodeType()    // Restituisce un valore costante (1, 2, 3, ..., 12) che
                  corrisponde a ELEMENT_NODE, ATTRIBUTE_NODE, TEXT_NODE, ..., NOTATION_NODE
getNodeValue()    //Restituisce il contenuto di un text node, il valore di un
                  attributo ecc., se applicato ad Element restituisce null
getOwnerDocument()
getParentNode()
hasChildNodes()
getChildNodes()
getFirstChild()
getLastChild()
getPreviousSibling()
getNextSibling()
hasAttributes()
getAttributes()
appendChild(newChild)
insertBefore(newChild, refChild)
replaceChild(newChild, oldChild)
removeChild(oldChild)
```

- `Document`

```
getDocumentElement()
createAttribute(name)
createElement(tagName)
createTextNode(data)
getDocType()
getElementById(IdVal)
```

- `Element`

```
getTagName()
getAttributeNode(name)
setAttributeNode(attr)
setAttribute(name, value)
getAttribute(name)
removeAttribute(name)
getElementsByTagName(name) // wild-card '*' per indicare tutti gli elementi
hasAttribute(name)
```

# Modelli web

## Modello statico

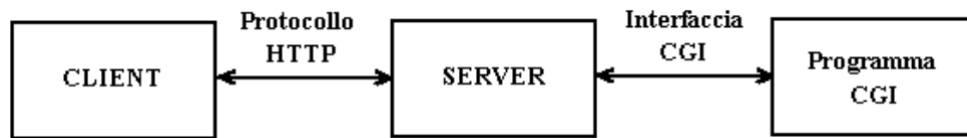
È un modello semplice, potente, di facile implementazione efficiente, ma presenta evidenti limiti

## Modello dinamico

### CGI (Common Gateway Interface)



La prima soluzione proposta per risolvere i limiti dati dal modello statico prende il nome di Common Gateway Interface (CGI):

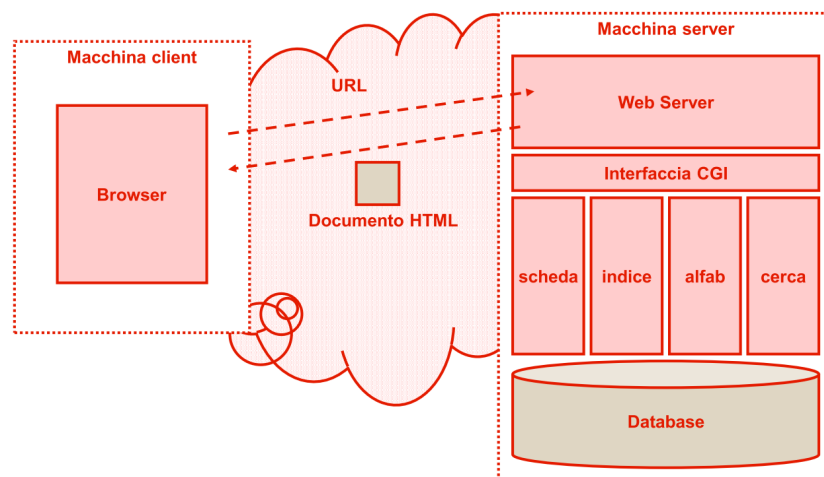


Se arriva ad esempio un URL `www.dino.it/cgi-bin/cerca` il server deve rendersi conto che `cerca` non è un documento HTML ma un programma CGI (di solito si sceglie come identificatore la cartella `/cgi-bin/`).

I programmi CGI e il server comunicano in quattro modi (specificati nell'interfaccia standard CGI):

- *Variabili di ambiente* del sistema operativo (REQUEST\_METHOD, QUERY\_STRING, REMOTE\_HOST, CONTENT\_TYPE, CONTENT\_LENGTH, HTTP\_USER\_AGENT)
- *Parametri* sulla linea di comandi: programma CGI viene lanciato in un processo pesante (si pensi a shell di sistema operativo che interpreta i parametri passati, ad esempio in metodo GET). La lunghezza massima della stringa dei parametri è 256 caratteri.
- *Standard Input* (usato con il metodo POST)
- *Standard Output*: per restituire al server la pagina HTML da inviare al client

Per rendere la struttura più dinamica le informazioni vengono memorizzate in un database e le applicazioni CGI si occupano di costruire il documento html basandosi sulle tabelle:



Problematiche legate all'utilizzo di CGI:

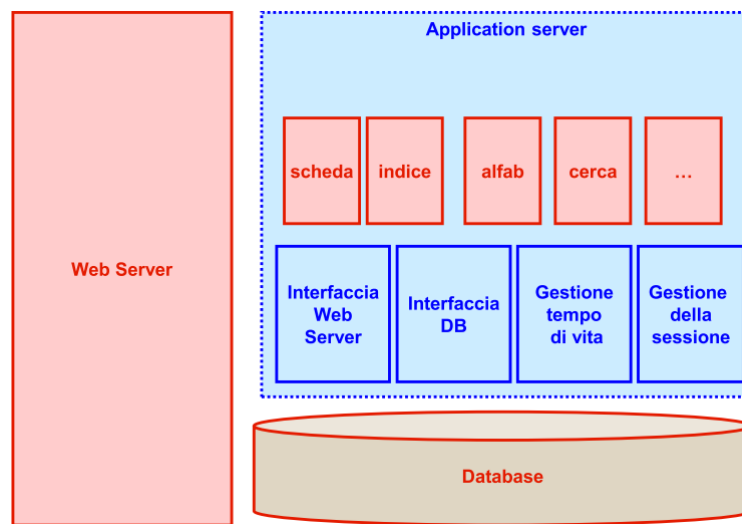
- Poca robustezza in C (errori bloccanti)
- Ogni programma CGI deve reimplementare tutta una serie di parti comuni (mancanza di moduli di base accessibili a tutti i programmi lato server): accesso al DB, logica di interpretazione delle richieste HTTP e di costruzione delle risposte, gestione dello stato ecc.
- Scarse garanzie sulla sicurezza

Soluzione: **Application server**

La soluzione migliore è quella di realizzare un contenitore che si preoccupa di fornire i servizi di cui le applicazioni hanno bisogno:

- interfacciamento con il Web Server
- gestione del tempo di vita (attivazione on-demand delle funzioni) e interfacciamento con il database
- supporto a QoS (tolleranza ai guasti)
- gestione della sicurezza.

Si ha così un'interfaccia modulare perchè composta da contenitore e applicazioni che svolgono operazioni semplificate.



## Stato

Parlando di applicazioni Web è possibile classificare lo stato in modo più preciso:

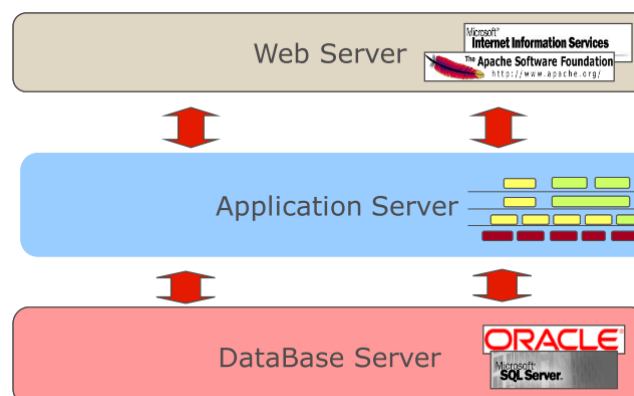
- Stato di *esecuzione* (insieme dei dati parziali per una elaborazione): rappresenta un avanzamento in una esecuzione; per sua natura è uno stato volatile; può essere mantenuto in memoria lato server come stato di uno o più oggetti
- Stato di *sessione* (insieme dei dati che caratterizzano una interazione con uno specifico utente): la sessione viene gestita di solito in modo unificato attraverso l'uso di istanze di oggetti specifici (supporto a oggetti sessione)

Lo stato di sessione deve essere condiviso da Client e Server e associato a una o più conversazioni effettuate da un singolo utente

Ci sono due tecniche di base per gestire lo stato, non necessariamente alternative ma integrabili:

- Utilizzo del meccanismo dei cookie (storage lato cliente)
- Gestione di uno stato sul server per ogni utente collegato (sessione server-side)
- Stato *informativo persistente* (ad esempio gli ordini inseriti da un sistema di eCommerce): viene normalmente mantenuto in una struttura persistente come un database

## Architettura frequente nei sistemi web



## Distribuzione verticale e orizzontale

### Replicazione: Web server

Web server è *stateless* per la natura del protocollo HTTP; per questo, molto facile da replicare e si possono applicare politiche di *load balancing* con diverse euristiche (usando dispositivi appositi oppure usando software come ad esempio il modulo apache mod\_ik).

### Replicazione: applicazione

La replicazione a livello di application server può risultare problematica, questo perché vengono spesso utilizzati oggetti o componenti con stato per motivi di performance (cache) o altre necessità specifiche.

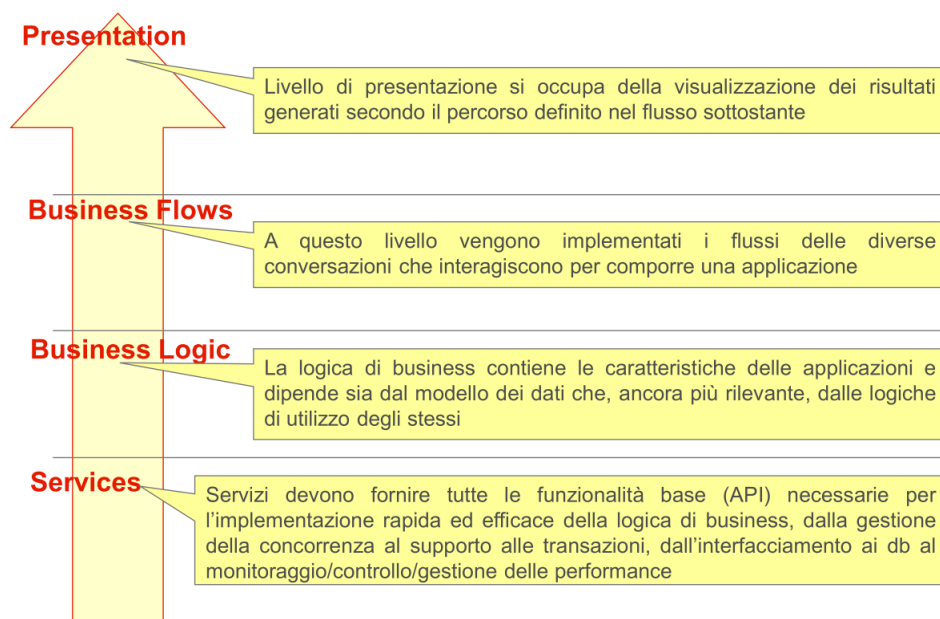
Se si mantiene lo stato concentrato all'interno della sessione e la sessione viene gestita interamente attraverso cookie, è possibile realizzare un framework applicativo completamente stateless lato server, ottenendo così realizzazione più semplice e primitiva replicabile in modo orizzontale.

### Replicazione: database

Il database server è (normalmente) un server stateful, la replicazione perciò è molto delicata perché deve mantenere il principio di atomicità delle transazioni

I database commerciali, come Oracle e Microsoft SQL Server prevedono delle configurazioni di clustering in grado di gestire in modo trasparente un numero variabile di CPU e macchine distinte (comunque in basso numero di unità)

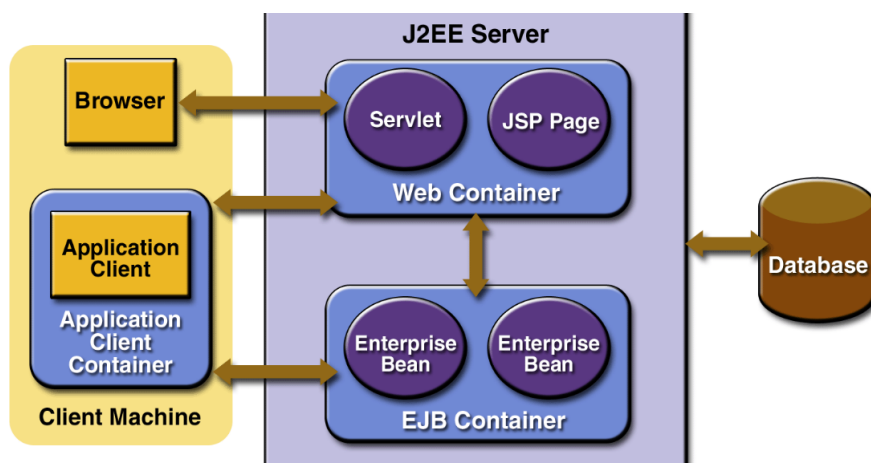
### Struttura multilivello



Non tutte le tecnologie permettono di rispettare questa suddivisione: in molti casi i sistemi vengono realizzati a 2 o 3 livelli.

Queste semplificazioni portano in certi casi a miglioramenti nelle performance e nella rapidità di sviluppo, ma possono comportare riduzione della leggibilità e della manutenibilità (va deciso in base al contesto di utilizzo se preferire l'una o l'altra).

## Architettura Java J2EE



Una **Web Application** è un gruppo di risorse server-side che nel loro insieme creano una applicazione interattiva fruibile via Web.

Le risorse server-side includono:

- Classi server-side (Servlet e classi standard Java)
- Java Server Pages
- Risorse statiche (documenti HTML, immagini, css, ...)
- Applet, Javascript e/o altri componenti che diventeranno attivi client-side
- Informazioni di configurazione e deployment

## Servlet

Una **Servlet** è una classe che fornisce un servizio comunicando con il client mediante protocolli request/response (il più diffuso è HTTP).

```
import javax.servlet.http.*;
public class HelloServlet extends HttpServlet{
    //All'arrivo di una richiesta HTTP il Servlet Container crea un oggetto request
    e
    //un oggetto response e li passa alla servlet
    public void doGet(HttpServletRequest request, HttpServletResponse response){
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<title>Hello World!</title>");
    }
}
```

Il **servlet container** controlla e supporta automaticamente il ciclo di vita di una servlet.

Se non esiste una istanza della servlet nel container:

- Carica la classe della servlet
- Crea una istanza della servlet
- Inizializza la servlet (invoca il metodo `init()`)

Poi, a regime:

- Invoca `service()` che identifica il tipo di richiesta ricevuta e chiama `doGet()` o `doPost()` a seconda del tipo di richiesta ricevuta passando come parametri due oggetti di tipo `HttpServletRequest` e `HttpServletResponse`
- Invoca `destroy()` quando la servlet deve essere disattivata

Normalmente più thread (client che effettuano richiesta al server) condividono la stessa istanza di una servlet. E' quindi necessario gestire le sezioni critiche utilizzando blocchi `synchronized`, semafori. Il modello `single-thread` è ormai deprecato e se si vuole far operare una servlet in tale modalità è necessario implementare l'interfaccia marker `SingleThreadModel`.

Alcuni metodi utili per la creazione della response:

```
void setStatus(int statusCode) //200 OK, 404 Page not found, ...
void sendError(int sc)
void sendError(int code, String message)
void setHeader(String headerName, String headerValue) //addHeader
void setDateHeader(String name, long millisecs) //addDateHeader
void setIntHeader(String name, int headerValue) //addIntHeader
void setContentType
void setContentLength
void addCookie(Cookie c)
void sendRedirect
PrintWriter getWriter()
ServletOutputStream getOutputStream()
```

Alcuni metodi per accedere alla request:

```
String getParameter(String parName)
String getContextPath()
String getQueryString()
String getPathInfo()
String getPathTranslated()
String getHeader(String name)
Enumeration getHeaders(String name)
int getIntHeader(String name)
long getDateHeader(String name)
String getRemoteUser()
String getAuthType()
boolean isUserInRole(String role)
Cookie[] getCookies()
InputStream getInputStream()
```

## Deployment

Un'applicazione Web deve essere installata e questo processo prende il nome di **deployment**. Il deployment comprende:

- La definizione del runtime environment di una Web Application
- La mappatura delle URL sulle servlet
- La definizione delle impostazioni di default di un'applicazione, ad es. welcome page e pagine di errore
- La configurazione delle caratteristiche di sicurezza dell'applicazione

Per definire la modalità con cui avviene il deployment dell'applicazione si utilizzano archivi .war (**Web Archives**) che sono di fatto file jar.

*web.xml* è un file di configurazione (in formato XML) che descrive la struttura dell'applicazione Web. Esso contiene l'elenco delle servlet e per ogni servlet permette di definire: nome, classe Java corrispondente, una serie di parametri di configurazione (coppie nome-valore, valori di inizializzazione).

E' il file che contiene mappatura fra URL e servlet che compongono l'applicazione.

## Servlet context

Ogni Web application esegue in un contesto: corrispondenza 1:1 tra una Web-app e suo contesto. L'interfaccia *ServletContext* è la vista della Web application (del suo contesto) da parte della servlet. Si può ottenere un'istanza di tipo *ServletContext* all'interno della servlet utilizzando il metodo `getServletContext()`.

## Attributi di contesto

Gli attributi di contesto sono accessibili a tutte le servlet e funzionano come variabili "globali".

```
//scrittura
ServletContext ctx = getServletContext();
ctx.setAttribute("utente1", new User("Giorgio Bianchi"));

//lettura
ServletContext ctx = getServletContext();
Enumeration aNames = ctx.getAttributeNames();
while (aNames.hasMoreElements()) {
    String aName = (String) aNames.nextElement();
    User user = (User) ctx.getAttribute(aName);
    ctx.removeAttribute(aName);
}
```

## Cookies

Un cookie contiene un certo numero di informazioni, tra cui:

- una coppia nome/valore
- il dominio Internet dell'applicazione che ne fa uso

- path dell'applicazione
- una expiration date espressa in secondi (-1 indica che il cookie non sarà memorizzato su file associato)
- un valore booleano per definirne il livello di sicurezza

La classe *Cookie* modella il cookie HTTP.

```
//creazione
Cookie c = new Cookie("MyCookie", "test");
c.setSecure(true);
c.setMaxAge(-1);
c.setPath("/");
response.addCookie(c);

//lettura
Cookie[] cookies = request.getCookies();
if(cookies != null) {
    for(int j=0; j<cookies.length j++) {
        Cookie c = cookies[j];
        out.println("Un cookie: " + c.getName()+"="+c.getValue());
    }
}
```

## Sessione

La sessione Web è un'entità gestita dal Web container . È condivisa fra tutte le richieste provenienti dallo stesso client: consente di mantenere, quindi, informazioni di stato (di sessione). Può contenere dati di varia natura ed è identificata in modo univoco da un **session ID**. Viene usata dai componenti di una Web application per mantenere lo stato del client durante le molteplici interazioni dell'utente con la Web application.

```
HttpSession session = request.getSession(true)

//gestione contenuto di una sessione
Cart sc = (Cart)session.getAttribute("shoppingCart");
sc.addItem(item);

session.setAttribute("shoppingCart", new Cart());
session.removeAttribute("shoppingCart");

Enumeration e = session.getAttributeNames();
while(e.hasMoreElements())
    out.println("Key; " + (String)e.nextElement());

//altre operazioni
String getId()
boolean isNew()
void invalidate()
long getCreationTime()
long getLastAccessedTime()
```

Il session ID è usato per identificare le richieste provenienti dallo stesso utente e mapparle sulla corrispondente sessione. Due tecniche per trasmettere l'id:

- includerlo in un cookie (session cookie): sappiamo però che non sempre i cookie sono attivati nel browser
- inclusione del session ID nella URL: si parla di **URL rewriting**

## Scope

Gli oggetti di tipo *ServletContext*, *HttpSession*, *HttpServletRequest* forniscono metodi per immagazzinare e ritrovare oggetti nei loro rispettivi ambiti (scope). Lo scope è definito dal tempo di vita (lifespan) e dall'accessibilità da parte delle servlet.

Ambito	Interfaccia	Tempo di vita	Accessibilità
<b>Request</b>	HttpServletRequest	Fino all'invio della risposta	Servlet corrente e ogni altra pagina inclusa o in forward
<b>Session</b>	HttpSession	Lo stesso della sessione utente	Ogni richiesta dello stesso client
<b>Application</b>	ServletContext	Lo stesso dell'applicazione	Ogni richiesta alla stessa Web app anche da clienti diversi e per servlet diverse

Gli oggetti scoped forniscono i seguenti metodi per immagazzinare e ritrovare oggetti nei rispettivi ambiti (scope):

```
void setAttribute(String name, Object o)
Object getAttribute(String name)
void removeAttribute(String name)
Enumeration getAttributeNames()
```

## Inclusione

Includere risorse Web (altre pagine, statiche o dinamiche) può essere utile quando si vogliono aggiungere contenuti creati da un'altra risorsa:

- Inclusione di risorsa statica: includiamo un'altra pagina nella nostra (ad es. banner)
- Inclusione di risorsa dinamica: la servlet inoltra una request ad un componente Web che la elabora e restituisce il risultato. Il risultato viene incluso nella pagina prodotta dalla servlet.

Per includere una risorsa si ricorre a un oggetto di tipo *RequestDispatcher* che può essere richiesto al contesto indicando la risorsa da includere:

```
RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher("/inServlet");
dispatcher.include(request, response);
```

## Forward

Si usa in situazioni in cui una servlet si occupa di parte dell'elaborazione della richiesta e delega a qualcun altro la gestione della risposta.

*In questo caso la risposta è di competenza esclusiva della risorsa che riceve l'inoltro*

Anche in questo caso si deve ottenere un oggetto di tipo *RequestDispatcher* da request passando come parametro il nome della risorsa:

```
RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher("/inServlet");
dispatcher.forward(request, response);
```

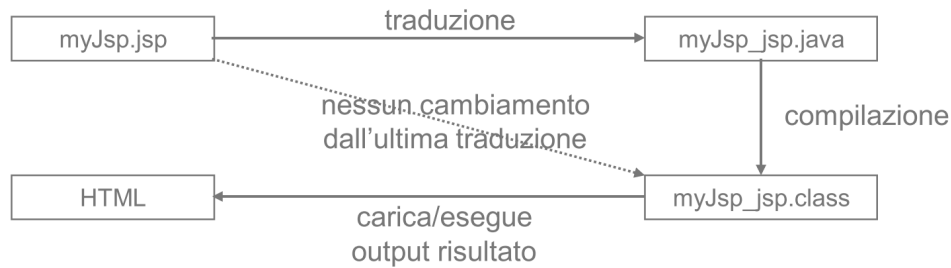
## JSP (Java Server Pages)

Le JSP sono l'altra componente base della tecnologia J2EE, relativamente alla parte Web. In realtà esse vengono trasformate in servlet dal container.

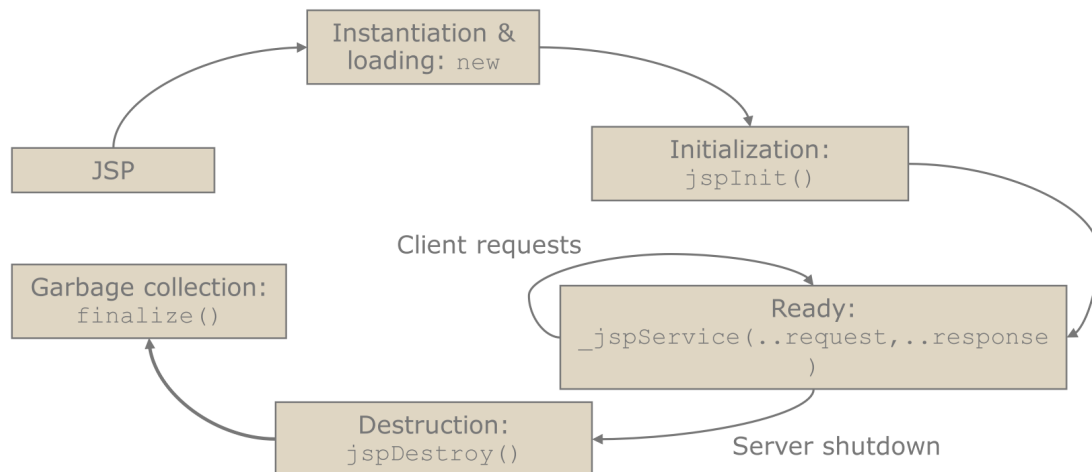
Le JSP non rendono inutili le servlet, rendono viceversa molto semplice presentare documenti HTML o XML (o loro parti) all'utente ma come tutti i linguaggi di script generano maggiori problemi di controllo della correttezza e testing.

Le richieste verso JSP sono gestite da una particolare servlet (in Tomcat si chiama *JspServlet*) che effettua le seguenti operazioni:

- traduzione della JSP in una servlet
- compilazione della servlet risultante in una classe
- esecuzione della JSP



Il ciclo di vita è:



## Tag

Le parti variabili della pagina sono contenute all'interno di tag speciali. Sono possibili due tipi di sintassi per questi tag:

- **Scripting-oriented tag**
  - Dichiarazione `<%! %>`
  - Espressione `<%= %>`
  - Scriptlet `<% %>`
  - Direttiva `<%@ %>`
- **XML-oriented tag**
  - Dichiarazione `<jsp:declaration>declaration</jsp:declaration>`
  - Espressione `<jsp:expression>expression</jsp: expression>`
  - Scriptlet `<jsp:scriptlet>java_code</jsp:scriptlet>`
  - Direttiva `<jsp:directive.dir_type dir_attribute />`

**Dichiarazione** : Si usano i delimitatori di dichiarazione per dichiarare variabili e metodi.

```

<%! String name = "Paolo Rossi";
double[] prices = {1.5, 76.8, 21.5};
double getTotal() {
    double total = 0.0;
    for (int i=0; i<prices.length; i++)
        total += prices[i];
    return total;
}
%>
  
```

**Espressione** : Si usano i delimitatori di espressione per valutare espressioni il cui risultato viene convertito in stringa e inserito nella pagina al posto del tag.



```
<p>Sig. <%=name%>,</p>
<p>l'ammontare del suo acquisto è: <%=getTotal()%> euro.</p>
<p>La data di oggi è: <%=new Date()%></p>
```

**Scriptlet** : Si usano i delimitatori di scriptlet per aggiungere un frammento di codice Java eseguibile dalla JSP.

```
<% if (userIsLogged) { %>
<h1>Benvenuto Sig. <%=name%></h1>
<% } else { %>
<h1>Per accedere al sito devi fare il login</h1>
<% } %>
```

**Direttiva** : Sono i comandi JSP valutati a tempo di compilazione e non producono output visibile. Le più importanti sono:

- **page** : definisce una serie di attributi che si applicano all'intera pagina

```
<%@ page
[ language="java" ]
[ extends="package.class" ]
[ import="{package.class | package.*}, ..." ]
[ session="true | false" ]
[ buffer="none | 8kb | sizekb" ]
[ autoFlush="true | false" ]
[ isThreadSafe="true | false" ]
[ info="text" ]
[ errorPage="relativeURL" ]
[ contentType="mimeType [ ;charset=characterSet ]" | "text/html ;
charset=ISO-8859-1" ]
[ isErrorPage="true | false" ]
%>
```

- **include** : include un altro documento

```
<%@ include file="myHeaderFile.html" %>
```

- **taglib** : carica una libreria di custom tag

```
<%@ uri="tagLibraryURI" prefix="tagPrefix"%>
```

## Built-in objects

Le specifiche JSP definiscono 9 oggetti built-in (o impliciti) utilizzabili senza dover creare istanze.

**page** `javax.servlet.jsp.HttpJspPage`

L'oggetto `page` rappresenta l'istanza corrente della servlet e può essere usato per accedere a tutti i metodi definiti nelle servlet

```
<%@ page info="Esempio di uso page." %>
<p>Page info: <%=page.getServletInfo()%></p>
```

**config** `javax.servlet.ServletConfig`

L'oggetto `config` contiene la configurazione della servlet (parametri di inizializzazione)

```
String getInitParameterName()           //Restituisce tutti i nomi dei parametri
di inizializzazione
String getInitParameter(String name)    //Restituisce il valore del parametro dato
il nome
```

**request** : `javax.servlet.http.HttpServletRequest`

L'oggetto request rappresenta la richiesta alla pagina JSP (è il parametro request passato al metodo service() della servlet)

```
String getParameter(String name)           // Restituisce il valore di un parametro
individuato dal nome
Enumeration getParameterNames()            // Restituisce l'elenco dei nomi dei
parametri
String getHeader(String name)              // Restituisce il valore di un header
individuato dal nome
Enumeration getHeaderNames()               // Restituisce l'elenco dei nomi degli
headers
Cookie[] getCookies()                      // Restituisce un array di oggetti cookie
che client ha inviato alla request
```

**response** : `javax.servlet.http.HttpServletResponse`

L'oggetto response rappresenta la risposta che viene restituita al client

```
void setHeader(String headerName, String headerValue) // Imposta header
void setDateHeader(String name, long millisecs)       // Imposta data
addHeader, addDateHeader, addIntHeader                // Aggiungono nuova
occorrenza di un dato header
setContentType                                       // Determina content-type
addCookie                                           // Consente di gestire i
cookie nella risposta
PrintWriter getWriter()                             // Restituisce uno stream
di caratteri (un'istanza di PrintWriter)
ServletOutputStream getOutputStream()                // Restituisce uno stream
di byte (un'istanza di ServletOutputStream)
```

**out** : `javax.servlet.jsp.JspWriter`

L'oggetto out è uno stream di caratteri e rappresenta lo stream di output della pagina

```
void print(String str)           // Stampa la stringa sullo stream di output
boolean isAutoFlush()            // Dice se output buffer è stato impostato in modalità
autoFlush o meno
getBufferSize()                  // Restituisce le dimensioni del buffer
getRemaining()                   // Indica quanti byte liberi ci sono nel buffer
void clearBuffer()               // Ripulisce il buffer
void flush()                     // Forza l'emissione del contenuto del buffer
void close()                     // Fa flush e chiude stream
```

**session** : `javax.servlet.http.HttpSession`

L'oggetto session fornisce informazioni sul contesto di esecuzione della JSP in termini di sessione utente

```
void setAttribute(String name, attributo) //
void setMaxInactiveInterval(int interval) //
String getID()                             // Restituisce ID di una sessione
boolean isNew()                             // Dice se sessione è nuova
void invalidate()                           // Permette di invalidare (distruggere)
una sessione
long getCreationTime()                      // Ci dice da quanto tempo è attiva la
sessione (in ms)
long getLastAccessedTime()                  // Ci dice quando è stata utilizzata
l'ultima volta
```

**application** : `javax.servlet.ServletContext`

L'oggetto application fornisce informazioni sul contesto di esecuzione della JSP con scope di visibilità comune a tutti gli utenti (è ServletContext).

Rappresenta la Web application a cui JSP appartiene e consente di interagire con l'ambiente di esecuzione:

- fornisce la versione di JSP Container
- garantisce l'accesso a risorse server-side
- permette accesso ai parametri di inizializzazione relativi all'applicazione
- consente di gestire gli attributi di un'applicazione

**pageContext** : `javax.servlet.jsp.PageContext`

L'oggetto pageContext fornisce informazioni sul contesto di esecuzione della pagina JSP. Rappresenta l'insieme degli oggetti built-in di una JSP. Consente l'accesso a tutti gli oggetti impliciti e ai loro attributi. Consente il trasferimento del controllo ad altre pagine.

Viene poco usato in caso di scripting, è più utile per costruire custom tag.

**exception** : `Java.lang.Throwable`

L'oggetto exception rappresenta l'eccezione che non viene gestita da nessun blocco catch. Non è disponibile in tutte le pagine ma solo nelle Error Page (quelle dichiarate con l'attributo `errorPage` impostato a true)

```
<%@ page isErrorPage="true" %>
<h1>Attenzione!</h1>
E' stato rilevato il seguente errore:
<br/> <b><%= exception %></b><br/>
<% exception.printStackTrace(out); %>
```

## Azioni

Le azioni sono comandi JSP tipicamente per l'interazione con altre pagine JSP, servlet, o componenti JavaBean; sono espresse usando sintassi XML:

### useBean

Un bean, non è altro che una classe Java dotata di alcune caratteristiche particolari: classe public, un costruttore public di default (senza argomenti), espone proprietà, sotto forma di coppie di metodi di accesso (accessors), espone eventi con metodi di registrazione che seguono regole precise.

```
<jsp:useBean id="user" class="RegisteredUser" scope="session"/>
```

Un oggetto bean si definisce come sopra e lo si può referenziare attraverso l'id. Lo scope può essere:

- page (default): fino a quando la pagina viene completata o fino al forward
- request: fino alla fine dell'elaborazione della richiesta e restituzione della risposta
- session: tempo di vita della sessione
- application: tempo di vita dell'applicazione

### getProperty

Consente l'accesso alle proprietà del bean

```
<jsp:getProperty name="beanId" property="propName"/>
```

### setProperty

Consente di modificare il valore della proprietà del bean

```
<jsp:setProperty name="beanId" property="propName" value="propValue"/>
```

## include

Consente di includere il contenuto generato dinamicamente da un'altra pagina locale all'interno dell'output della pagina corrente

```
<jsp:include page="localURL" flush="true">
  <jsp:param name="parName1" value="parValue1"/>
  <jsp:param name="parNameN" value="parValueN"/>
</jsp:include>
```

## forward

Consente il trasferimento del controllo dalla pagina JSP corrente ad un'altra pagina sul server locale

Attenzione: *forward* è possibile soltanto se non è stato emesso alcun output

```
<jsp:forward page="localURL" />
<!-- E' possibile generare dinamicamente attributo page -->
<jsp:forward page='<%= "message"+statusCode+".html"%>' />

<!-- E' possibile specificare parametri -->
<jsp:forward page="localURL">
  <jsp:param name="parName1" value="parValue1"/>
  <jsp:param name="parNameN" value="parValueN"/>
</jsp:forward>
```

## plugin

Genera contenuto per scaricare plug-in Java se necessario

## Custom tag

JSP permettono di definire tag personalizzati (custom tag) che estendono quelli predefiniti. Una **taglib** è una collezione di questi tag non standard, realizzata mediante una classe Java:

```
<!-- La direttiva per includere la libreria di tag -->
<%@ uri="tagLibraryURI" prefix="tagPrefix"%>

<!-- Utilizzo: il prefisso definisce un namespace e quindi elimina eventuali
omonimie date dall'inclusione da più librerie-->
<html:helloWorld who="Mario">
```

# Javascript

JavaScript è un linguaggio **interpretato** e non compilato, è **object-based** ma non **class-based** cioè esiste il concetto di oggetto ma non esiste il concetto di classe. Inoltre JavaScript è **debolmente tipizzato** ossia non è necessario definire il tipo di una variabile.

Javascript come Java ha una sintassi modellata su quella del C:

- è un linguaggio case-sensitive
- le istruzioni sono terminate da ; ma il terminatore può essere omissso se si va a capo
- sono ammessi sia commenti multilinea (delimitati da /\* e \*/) che mono-linea (iniziano con //)
- gli identificatori possono contenere lettere, cifre e i caratteri \_ e \$ ma non possono iniziare con una cifra

Il codice JavaScript viene eseguito da un interprete contenuto all'interno del browser.

Uno script Javascript viene inserito nella pagina HTML usando il tag `<script>`

```
<html>
  <body>
    <p>Hello da JavaScript</p>
    <script type="text/javascript">
      alert("Hello World!");      // Esistono anche pop-up di tipo confirm e
prompt
    </script>
  </body>
</html>
```

## Variabili

Le variabili vengono dichiarate usando la parola chiave var:

```
var nomevariabile;
```

i tipi primitivi ammessi sono:

- numeri (number) : rappresentati sempre in formato floating point, esiste il valore NaN e infinite.
- booleani (boolean)

i tipi riferimento sono:

- oggetti
- array

Esiste lo scope globale e quello locale (ovvero dentro una funzione) ma, a differenza di Java, non esiste lo scope di blocco.

Ad ogni variabile può essere assegnato il valore `null` che rappresenta l'assenza di un valore, una variabile non inizializzata ha invece un valore indefinito `undefined`. I due concetti si assomigliano ma non sono uguali.

## Oggetti

Gli oggetti sono tipi composti che contengono un certo numero di proprietà (attributi). Tali proprietà non sono definite a priori ma possono essere aggiunte dinamicamente.

```
var o = new Object();
o.x = 7;
o.y = 8;
o.tot = o.x + o.y;
alert(o.tot);
```

*Object() è un costruttore e non una classe. Le classi non esistono e quindi i due concetti non si sovrappongono come avviene in Java!*

Le costanti oggetto (**object literal**) sono racchiuse fra parentesi graffe e contengono un elenco di attributi nella forma: `nome:valore`

```
var o = {x:7, y:8, tot:15};
alert(o.tot);
```

## Array

Gli array sono tipi composti i cui elementi sono accessibili mediante un indice numerico (l'indice parte da zero) e non hanno una dimensione prefissata (simili agli ArrayList di Java).

```
var arr = new Array([dimensione]); // Dimensione è opzionale
//oppure
var arr = [val, val2, ..., valn];
```

## Stringhe

Mentre in Java le stringhe sono oggetti che sembrano dati di tipo primitivo in JavaScript sono dati di tipo primitivo che sembrano oggetti (questo perchè sono gestite mediante classe wrapper String quando la variabile necessita di essere trattata come tipo riferimento)

Esiste la possibilità di definire le stringhe delimitate da apici singoli o doppi. E' possibile la concatenazione con l'operatore + ed è possibile la comparazione con gli operatori di uguaglianza, disuguaglianza, maggioranza e minoranza.

## Regular expressions

JavaScript ha un supporto per le espressioni regolari (regular expressions) che sono un tipo di dato nativo del linguaggio

```
var r = /[abc]/;  
var r = new RegExp("[abc]");
```

## Funzioni

Una funzione è un frammento di codice JavaScript che viene definito una volta e usato in più punti, ammette parametri che sono privi di tipo e restituisce un valore il cui tipo non è definito.

```
function sum(x,y) {  
    return x+y;  
}  
  
//function literal  
var sum = function(x,y) { return x+y; }  
  
//function object  
var sum = new Function("x", "y", "return x+y;");
```

Quando una funzione viene assegnata ad una proprietà di un oggetto viene chiamata **metodo** dell'oggetto

```
var o = new Object();  
o.x = 7;  
o.y = 8;  
o.tot = function() { return this.x + this.y; }  
alert(o.tot());
```

## Costruttori

Un costruttore è una funzione che ha come scopo quello di costruire un oggetto. L'oggetto da costruire è accessibile con la parola chiave `this`.

```
function Rectangle(w, h) {  
    this.w = w;  
    this.h = h;  
    this.area = function() { return this.w * this.h; }  
    this.perimeter = function() { return 2*(this.w + this.h); }  
}
```

## Operatori

JavaScript ammette tutti gli operatori presenti in C e in Java. Esistono poi alcuni operatori tipici:

- `delete` : elimina una proprietà di un oggetto
- `void` : valuta un'espressione senza restituire alcun valore
- `typeof` : restituisce il valore di un operando
- `===` : identità o uguaglianza stretta (diverso da `==` che verifica l'uguaglianza)
- `!==` : non identità (diverso da `!=`)

## Inserimento codice javascript in una pagina HTML

- HTML prevede un apposito tag per inserire script; la sua sintassi è

```
<!-- script interno -->  
<script type="application/javascript"> <!-- script-text //--> </script>
```

Il commento che racchiude il testo dello script serve per gestire la compatibilità con i browser che non gestiscono JavaScript (in quei casi viene ignorato).

```
<!-- script esterno -->  
<script language="Javascript" src="nomefile.js"> </script>
```

## Costruzione dinamica della pagina

La più semplice modalità di utilizzo di JavaScript consiste nell'inserire nel corpo della pagina script che generano dinamicamente parti della pagina HTML.

Bisogna tener presente che questi script vengono eseguiti solo una volta durante il caricamento della pagina e quindi non si ha interattività con l'utente.

L'uso più comune è quello di generare pagine diverse in base al tipo di browser o alla risoluzione dello schermo.

La pagina corrente è rappresentata dall'oggetto **document** e per scrivere nella pagina si utilizzano `document.write()` o `document.writeln()`.

Per accedere alle informazioni del browser si utilizza l'oggetto **navigator**:

```
navigator.appCodeName //nome in codice del browser (poco utile)  
navigator.appName     // Nome del browser  
navigator.appVersion  // Versione del browser  
navigator.cookieEnabled // Dice se i cookies sono abilitati  
navigator.platform    // Piattaforma per cui il browser è stato compilato (es.  
Win32)  
navigator.userAgent   // Stringa passata dal browser come header user-agent (es.  
"Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1;)")
```

L'oggetto **screen** permette di ricavare informazioni sullo schermo:

```
screen.width  
screen.height
```

## Modello ad eventi e interattività

Per avere una reale interattività bisogna utilizzare il meccanismo degli eventi.

JavaScript consente di associare script agli eventi causati dall'interazione dell'utente con la pagina HTML. L'associazione avviene mediante attributi collegati agli elementi della pagina HTML. Gli script prendono il nome di gestori di eventi.

## Ajax

Applicazioni Web tradizionali espongono un modello di interazione rigido (click, wait, and refresh) sincrono: l'utente effettua una richiesta e deve attendere la risposta da parte del server. Il modello di interazione AJAX è un'estensione di Javascript nato per superare queste limitazioni permettendo di definire un modello asincrono.

La tipica sequenza AJAX è:

1. Si verifica un evento determinato dall'interazione fra utente e pagina Web
2. L'evento comporta l'esecuzione di una funzione JavaScript in cui:

- Si istanzia un oggetto di classe XMLHttpRequest
  - Si configura XMLHttpRequest: si associa una funzione di callback, si effettua configurazione, ...
  - Si effettua chiamata asincrona al server
3. Il server elabora la richiesta e risponde al client
  4. Il browser invoca la funzione di callback che:
    - elabora il risultato
    - aggiorna il DOM della pagina per mostrare i risultati dell'elaborazione

## XMLHttpRequest

È l'oggetto che effettua richiesta di una risorsa via HTTP a server Web. Può effettuare richieste GET e POST. Le richieste possono essere sincrone (non ci interessano) o asincrone (thread dedicato).

*Nota: per motivi di sicurezza XmlHttpRequest può essere utilizzata solo verso dominio da cui proviene la risorsa che la utilizza*

### Metodi di invio della richiesta

```
var xhr = new XMLHttpRequest();

open(method, uri [, async] [, user] [,password])    // Permette di inizializzare la
richiesta da formulare al server

setRequest(header, valore)        // Consente di impostare gli header HTTP della
richiesta da inviare

send(body)        // Consente di inviare la richiesta al server
```

## Proprietà

### readyState

Proprietà in sola lettura di tipo intero che consente di leggere in ogni momento lo stato della richiesta. Ammette 5 valori:

- 0: uninitialized – l'oggetto esiste, ma non è stato ancora richiamato open()
- 1: open – è stato invocato il metodo open(), ma send() non ha ancora effettuato l'invio dati
- 2: sent – metodo send() è stato eseguito e ha effettuato la richiesta
- 3: receiving – la risposta ha cominciato ad arrivare • 4: loaded – l'operazione è stata completata

### onreadystatechange

Come si è detto l'esecuzione del codice non si blocca sulla send() in attesa dei risultati. Per gestire la risposta si deve quindi adottare un approccio a eventi. Occorre registrare una funzione di callback che viene richiamata in modo asincrono ad ogni cambio di stato della proprietà ReadyState:

```
xhr.onreadystatechange = nomefunzione
xhr.onreadystatechange = function() {istruzioni}
```

### status

Contiene un valore intero corrispondente al codice HTTP dell'esito della richiesta

### statusText

Contiene invece una descrizione testuale del codice HTTP restituito dal server

### responseText



Stringa che contiene il body della risposta HTTP disponibile solo a interazione ultimata (readyState==4)

### responseXML

Body della risposta convertito in documento XML (se possibile) al fine di permettere la navigazione attraverso javascript. Può essere null se i dati restituiti non sono un documento XML ben formato.

### Metodi per la ricezione della risposta

Sono utilizzabili solo nella funzione di callback e possono essere invocati in modo safe solo a richiesta conclusa (readyState==4):

```
getAllResponseHeaders()
getResponseHeader(header_name) // Consente di leggere gli header HTTP che
                                descrivono la risposta del server
```

Le richieste AJAX permettono all'utente di continuare a interagire con la pagina ma non necessariamente lo informano di che cosa stia succedendo e possono durare troppo! L'effetto è un possibile disorientamento dell'utente.

Di conseguenza, di solito si agisce su due fronti per limitare i comportamenti impropri a livello utente:

- Rendere visibile in qualche modo l'andamento della chiamata (barre di scorrimento, info utente, ...)
- Interrompere le richieste che non terminano in tempo utile per sovraccarichi del server o momentanei problemi di rete (timeout)

A tal proposito il metodo:

```
abort() //Consente l'interruzione delle operazioni di invio o ricezione e non
        va invocato nella callback
```

*Attenzione: non ha senso invocare `abort()` dentro la funzione di callback.*

*Si crea un'altra funzione da far richiamare in modo asincrono al sistema mediante il metodo `setTimeout(funzioneAsincronaPerAbortire,timeOut)`*

## JSON (JavaScript Object Notation)

JSON è un formato per lo scambio di dati, considerato molto più comodo di XML perché è leggero in termini di quantità di dati scambiati, semplice ed efficiente da elaborare da parte del supporto runtime al linguaggio di programmazione (in particolare per JavaScript) e ragionevolmente semplice da leggere per operatore umano.

```
var Beatles = {
  "Paese" : "Inghilterra",
  "AnnoFormazione" : 1959,
  "TipoMusica" : "Rock",
  "Membri" : ["Paul", "John", "George", "Ringo"]
}
```

Gli array possono contenere anche altri oggetti: la sintassi di un oggetto è un insieme di parametri racchiusi tra { }.

Javascript mette a disposizione la funzione `eval()` che invoca l'interprete per la traduzione della stringa passata come argomento.

```
var s = '{ "Paese" : "Inghilterra",
          "AnnoFormazione" : 1959,
          "TipoMusica" : "Rock",
          "Membri" : ["Paul", "John", "George" , "Ringo"] }';
var o = eval('(' + s + ')')      //La stringa da valutare deve essere posrta tra
parentesi tonde
o.Paese      //Inghilterra
```

Uso di eval() presenta rischi: stringa passata come parametro potrebbe contenere codice malevolo.

Di solito si preferisce utilizzare parser appositi che traducono solo oggetti JSON e non espressioni JavaScript di qualunque tipo. Alcuni parser molto diffusi:

- Google GSON - <https://github.com/google/gson>
- «più tradizionale» jabsorb - <https://github.com/Servoy/jabsorb>

## JSON e AJAX

Sul client:

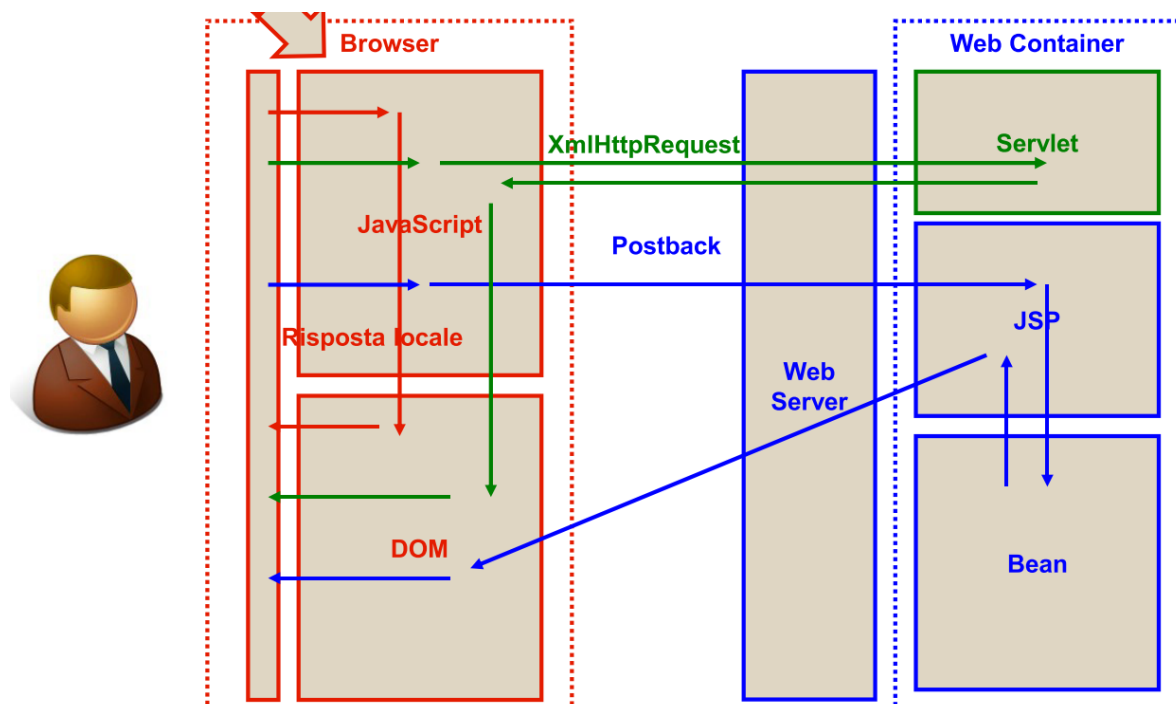
1. Si crea un oggetto JavaScript e si riempiono le sue proprietà con le informazioni necessarie
2. Si usa `JSON.stringify()` per convertire l'oggetto in stringa JSON
3. Si usa la funzione `encodeURIComponent()` per convertire la stringa in un formato utilizzabile in una richiesta HTTP
4. Si manda la stringa al server mediante `XMLHttpRequest` (stringa viene passata come variabile con GET o POST)

Sul server:

1. Si decodifica la stringa JSON e la si trasforma in oggetto Java utilizzando un apposito parser
2. Si elabora l'oggetto
3. Si crea un nuovo oggetto Java che contiene dati della risposta
4. Si trasforma l'oggetto Java in stringa JSON usando il parser suddetto
5. Si trasmette la stringa JSON al client nel corpo della risposta HTTP:  
`response.out.write(strJSON);`

Sul client:

1. Si converte la stringa JSON in un oggetto Javascript usando `JSON.parse()`
2. Si usa liberamente l'oggetto per gli scopi desiderati



# Accesso diretto alle basi di dati

Due soluzioni possibili:

- SQL Embedded

L'incastonamento prevede di introdurre direttamente nel programma sorgente le istruzioni SQL distinguendole dalle normali istruzioni tramite un opportuno separatore (lo standard SQL prevede che il codice SQL sia preceduto dalla stringa `exec sql` e termin con `;`)

- Cursori

Strumento che permette a un programma di accedere alle righe di una tabella una alla volta

- SQL dinamico

Permette all'applicazione di definire al momento dell'esecuzione le interrogazioni SQL da effettuare sulla base dati

- Call Level Interface (CLI):

L'integrazione con SQL avviene tramite l'invocazione di una opportuna libreria di funzioni definita dal DBMS e per questo esso rappresenta uno strumento più flessibile e meglio integrato con il linguaggio di programmazione (ODBC, OLE DB, ADO e JDBC).

Un'applicazione che accede ad una sorgente di dati ha bisogno di fare le seguenti operazioni:

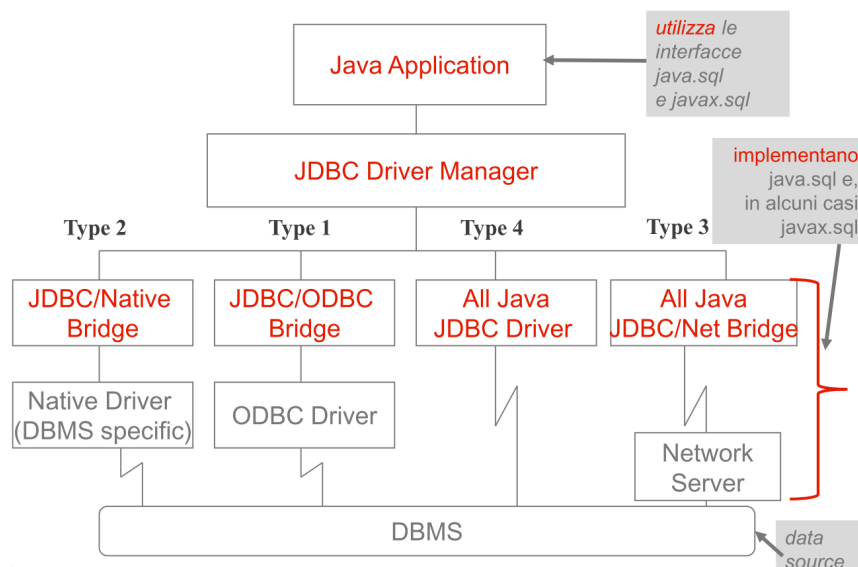
1. aprire una connessione alla sorgente dati
2. inviare attraverso la connessione istruzioni (di interrogazione e aggiornamento) alla sorgente dati
3. processare i risultati ricevuti dalla sorgente dati in risposta alle istruzioni inviate

## ODBC

API standard standard definita da Microsoft, acronimo di Open Database Connectivity, permette l'accesso a dati residenti in DB relazionali in un contesto eterogeneo e distribuito. Permette ai programmatori di formulare richieste SQL che accederanno a dati relativi a DB distinti senza dover conoscere le interfacce proprietarie di ogni singolo DB poiché gestisce richieste SQL convertendole in un formato comprensibile al particolare DBMS.

## JDBC

API Java standard definita da Sun Microsystem. Rappresenta la controparte Java di ODBC. Permette di accedere a database locali e remoti in modo uniforme in modo indipendente dalla piattaforma. Le classi Java che realizzano funzionalità JDBC sono contenute nei package `java.sql` e `javax.sql`.



```

Class.forName("com.ibm.db2.jcc.DB2Driver").newInstance();
Connection conn = DriverManager.getConnection("jdbc:db2:MYDB");
Statement stm = conn.createStatement();
ResultSet res = stm.executeQuery("SELECT * FROM MYTABLE");
while (res.next()) {
    String col1 = res.getString("MYCOL1");
    int col2 = res.getInt("MYCOL2");
}

```

## Interfaccia Driver

Rappresenta il punto di partenza per ottenere una connessione a un DBMS. I produttori di driver JDBC implementano l'interfaccia Driver (mediante opportuna classe) affinché possa funzionare con un tipo particolare di DBMS.

È possibile ottenere un'istanza effettiva della classe Driver ricercando la classe con il metodo `forName`:

```

Driver d = Class.forName("com.ibm.db2.jcc.DB2Driver").newInstance();

```

Ogni driver JDBC ha una stringa di connessione che riconosce nella forma

`jdbc:product_name:database_alias`. Nel caso della configurazione DB2 in laboratorio:

`jdbc:db2://diva.disi.unibo.it:50000/[sample|tw_stud]`

## Classe DriverManager

Quando un oggetto Driver viene istanziato, esso viene automaticamente registrato nella classe `DriverManager`. Consente la connessione con il DBMS sottostante mediante il metodo statico `getConnection()` e usa il driver opportuno tra quelli registrati.

## Interfaccia Connection

Un oggetto di tipo `Connection` rappresenta una connessione attiva con il DB.

## Interfaccia Statement

Gli oggetti di tipo `Statement` possono essere usati per inviare query SQL semplici (senza uso di parametri) verso il DBMS sottostante. Le query possono essere:

- UPDATE, INSERT, CREATE (`executeUpdate(stmt SQL)`)
- SELECT (`executeQuery(stmt SQL)`): il risultato è inserito in un oggetto `ResultSet`
- Un insieme di operazioni una dietro l'altra (`execute(stmt SQL)`)

Un oggetto `Statement` può essere creato con il metodo `createStatement` di `Connection`

## Interfaccia PreparedStatement

Gli oggetti di tipo `PreparedStatement` possono essere usati per creare query SQL parametriche (parametri di tipo IN) e precompilate ("prepared").

Il valore di ciascun parametro non è specificato nel momento in cui lo statement SQL è definito, ma rimpiazzato dal carattere '?'.  
 È più efficiente perché una volta che la query è preparata, sarà pre-compilata per usi futuri.

Un oggetto `PreparedStatement` può essere creato con il metodo `prepareStatement` di `Connection`.

## Interfaccia CallableStatement

Gli oggetti di tipo `CallableStatement` possono essere usati per definire query parametriche con parametri di tipo IN, OUT e INOUT.

Permettono di eseguire una invocazione a una **stored procedure** memorizzata sul server DB.

Un oggetto CallableStatement può essere creato con il metodo `prepareCall` di Connection.

## Interfaccia ResultSet

L'oggetto ResultSet è il risultato di una query di selezione (SELECT).

Rappresenta una tabella composta da righe (gli elementi selezionati) e colonne (gli attributi richiesti) esplorabile mediante **cursori**:

- per ottenere il valore relativo alla riga corrente attraverso gli accessor tipizzati  
`getXXX(column_name)` o `getXXX(column_number)`
- per spostare il cursore sulla riga successiva si usa `next()` che restituisce true in caso di successo altrimenti false

I valori NULL SQL sono convertiti in null, 0, o false, dipendentemente dal tipo di metodo `getXXX`.

Per determinare se un particolare valore di un risultato corrisponde a NULL in JDBC si usa il metodo `wasNull()`.

## JDBC e DB2

DB2 prevede 2 driver JDBC di tipo two-tier:

- Net-Driver: `com.ibm.db2.jcc.DB2Driver` per connettersi via rete (ad esempio, mediante il protocollo di rete TCP/IP) a server remoti a condizione che questi abbiano attivo il servizio DB2 JDBC Applet Server. E' un driver universale, ovvero indipendente dalla piattaforma.
- Driver Java "puro": `com.ibm.db2.jcc.DB2Driver` per connettersi ad istanze DB2 residenti sulla macchina locale o catalogate localmente

**Passi principali:**

1. Importazione package
2. Registrazione driver JDBC
3. Apertura connessione al DB (**Connection**)
4. Creazione oggetto **Statement**
5. Esecuzione query e eventuale restituzione oggetto **ResultSet**
6. Utilizzo risultati
7. Chiusura oggetto/i **ResultSet** e oggetto/i **Statement**
8. Chiusura connessione

```
//importazione package
import java.sql.*; //package JDBC

class Esempio {
    public static void main(String argv[]) {
        try {
            // caricamento e registrazione driver
            Class.forName("com.ibm.db2.jcc.DB2Driver").newInstance();
            Connection con = null; // URL jdbc:db2:database_alias
            String url = "jdbc:db2://diva.disi.unibo.it:50000/sample";
            if (argv.length == 2) {
                String userid = argv[0];
                String passwd = argv[1];
                // connessione con id/passwd forniti dall'utente
                con = DriverManager.getConnection(url, userid, passwd);
            } else {
                System.out.println("\nUsage: java Esempio username password\n");
                System.exit(0);
            }
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
            System.exit(1);
        }
    }
}
```

```

    }
    // interrogazione table EMPLOYEE
    System.out.println("Retrieve some data from the database");
    Statement stmt = con.createStatement();
    // esegue la query
    ResultSet rs = stmt.executeQuery("SELECT * FROM DB2INST1.EMPLOYEE");
    System.out.println("Received results:");
    // mostra i risultati
    //rs.next() = "false" se non ci sono più righe risultato
    while (rs.next()) {
        String number = rs.getString(1);
        String name = rs.getString(2);
        System.out.print(" empno= " + number);
        System.out.print(" firstname= " + name);
        System.out.print("\n");
    }
    // chiude ResultSet e Statement
    rs.close();
    stmt.close();
    // Esecuzione di altre istruzioni SQL // aggiorna il database... prova ad
aggiornarlo!
    System.out.println("\n\nUpdate the database... ");
    stmt = con.createStatement();
    int rowsUpdated = stmt.executeUpdate("UPDATE DB2INST1.EMPLOYEE SET
firstname = 'SHILI' WHERE empno = '000010'");
    System.out.print("Changed "+rowsUpdated);
    if (1 == rowsUpdated)
        System.out.println(" row.");
    else
        System.out.println(" rows.");
    // chiude Statement
    stmt.close();
    // chiude Connection
    con.close();
} catch( Exception e ) {
    e.printStackTrace();
}
}
}

```

## Conflitto di impedenza

Nel modello OO le relazioni tra oggetti sono realizzate con riferimenti mentre nel modello relazionale sono realizzate con i valori.

Nel mondo OO i dati sono identificati da OID (trasparenti al programmatore) mentre in SQL le ennuple sono identificate da chiavi primarie (visibili e manipolabili). L'ereditarietà non ha controparte nel mondo relazionale.

## Forza bruta

E' la tecnica più semplice per gestire la persistenza e determina un forte accoppiamento con la sorgente dati. Consiste nello scrivere dentro le classi del modello un insieme di metodi che implementano le operazioni CRUD (Create, Retrieve, Update, Delete).

Per ogni classe MyC che rappresenta una entità del dominio si definiscono:

- un metodo `doRetrieveByKey(X key)` che restituisce un oggetto data la chiave
- un metodo `saveOrUpdate()` che salva i dati dell'oggetto corrente
- uno o più metodi `doRetrieveByCond(...)` che restituiscono una collezione di oggetti che soddisfano una condizione
- un metodo `doDelete(...)` che cancella l'oggetto corrente

La soluzione forza bruta non è particolarmente conveniente perché l'accoppiamento tra la sorgente dati e le classi del modello è molto forte.

## Pattern DAO (Data Access Object)

Il pattern DAO è uno degli standard "J2EE design patterns" e rappresenta un modo possibile di separare logica di business da logica di persistenza. I componenti della logica di business non dovrebbero mai contenere codice che accede direttamente al database.

I valori scambiati tra DB e il resto dell'applicazione sono racchiusi in oggetti detti **DTO (Data Transfer Object)** che includono campi privati, getter/setter e metodi di utilità (questo per essere a tutti gli effetti java bean).

Le operazioni che coinvolgono tali oggetti sono raggruppate in interfacce che definiscono i DAO (Data Access Object). Le implementazioni di tali interfacce permettono l'effettivo accesso al database. Gli oggetti DAO non sono istanziati direttamente dai componenti ma possono essere ottenuti attraverso metodi factory.

Riassunto componenti:

- Una factory astratta unica che espone un metodo di creazione parametrico per ottenere factory concrete
- Una factory concreta per ogni tipo di DB supportato
- Un oggetto DTO per ogni tipo di entità che si vuole rappresentare (package modello)
- Un'interfaccia DAO per ogni oggetto DTO (package persistenza)
- Un'implementazione dell'interfaccia DAO di un DTO per ciascun DB supportato (package persistenza.db2)

## Modello retrieve : ricostruzione degli oggetti dal DB

Il problema principale nel ricostruire oggetti a partire dalla base di dati consiste nel ricostruire i riferimenti: chiave esterne vs riferimenti e join vs navigazioni.

Nel progettare le classi, i riferimenti vanno messi considerando la direzione in cui le corrispondenti associazioni sono navigate dalle operazioni del sistema.

Nel caso in cui non ha particolarmente senso costruire un oggetto completo si ricorre a una tecnica detta caricamento pigro:

### Lazy load

Permette di caricare i dati solo quando questi sono effettivamente richiesti. Questo significa che se il metodo non viene mai invocato, il costo di caricare i dati non viene mai pagato.

Per implementare bene la strategia Lazy Load senza ridurre la coesione nella classe del modello è utile fare riferimento al **pattern Proxy**, cioè la classe `xxxxDAOImpl` invece di restituire un bean completo restituirà un proxy associato. La classe proxy estende la classe bean con i metodi che devono implementare le operazioni onerose. I proxy appartengono al package persistenza.db2.

### Caricamento Eager

Per le classi con associazioni per le quali non conviene applicare la strategia Lazy Load ricostruiamo la rete (parziale) di oggetti con i risultati di un join.

## Modello create: rendere persistenti gli oggetti

Se non tutte le classi hanno identificatori naturali semplici è utile introdurre chiavi surrogate (un identificativo intero). La chiave viene creata al momento dell'inserimento di un oggetto nel DB. Soluzioni:

- Soluzione naive: chiedo al DB il valore massimo, lo incremento ed eseguo l'inserimento
- Uso campi auto-incrementanti: Dopo l'inserimento devo interrogare il DB per riottenere tale valore

- Uso una **sequenza SQL** che interrogo per farmi dare un nuovo id prima di ogni inserimento (soluzione più elegante, efficiente e semplice da implementare)

```
CREATE SEQUENCE sequence_id
AS INTEGER
START WITH 1
INCREMENT BY 1
MINVALUE 0
MAXVALUE 9999999
NO CYCLE;
```

La responsabilità della gestione degli id viene affidata ad una classe opportuna che chiamiamo IdBroker. Al solito si definisce un'interfaccia IdBroker le cui implementazioni sono DBMS-dependent.

## ORM (Object Relational Mapping)

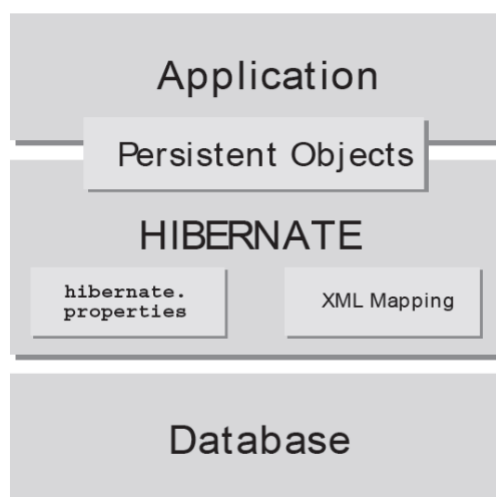
La tecnica ORM aumenta il livello di astrazione semplificando l'utilizzo allo sviluppatore ma aumenta la complessità avendo ripercussioni sulle performance.

ORM è la "persistenza automatica (e trasparente) di oggetti di applicazioni Java in tabelle di un DB relazionale, basata su metadati che descrivono il mapping tra oggetti e DB". Lavora trasformando (in modo reversibile) i dati da una rappresentazione all'altra.

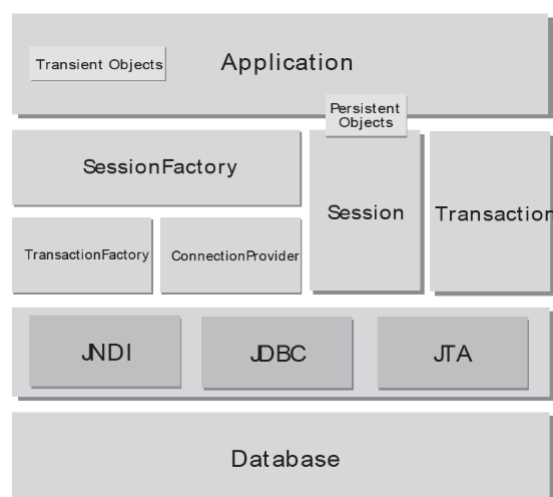
**Hibernate** è un tool full ORM. I suoi vantaggi sono:

- produttività: elimina la maggior parte della gestione della persistenza.
- manutenibilità: meno linee di codice, più inerenti agli aspetti di business logic che dettagli implementativi
- performance: persistenza hand-coded più performante rispetto a quella automatica (occorre però considerare aspetti di tempo e budget)
- indipendenza del vendor: ORM astrae una applicazione dal DB sottostante. Se un tool ORM supporta un certo numero di DB ciò conferisce portabilità all'applicazione

Il framework Hibernate rappresenta lo standard risolutivo per il mapping automatico tra oggetti Java e relazioni del DB. Tale mapping si può realizzare per mezzo di descrittori XML. Lo sviluppatore è esonerato da richieste di esecuzione di chiamate SQL e gestione manuale dei risultati. L'applicazione rimane portabile in tutti i sistemi di gestione supportati, con pochissimo overhead.



Architettura di alto livello



Architettura "completa"

## SessionFactory

Factory per oggetti **Session** e cliente di **ConnectionProvider**: una factory per ogni DB.

## Session



Rappresenta un contesto di persistenza (wrapper a JDBC **Connection**) e la sua vita è delimitata dall'inizio e dalla fine di una transazione logica.

Fa da factory per oggetti **Transaction**.

## Transaction

Oggetto single-threaded usato dall'applicazione per specificare unità atomiche di lavoro. Permette di astrarre dai dettagli dei sottostanti meccanismi transazionali.

## ConnectionProvider

Factory per un pool di connessioni JDBC. Astrae l'applicazione dal Data Source o dal Driver Manager sottostante. Non è esposta alla applicazione ma può essere estesa/implementata dallo sviluppatore.

## Transactionfactory

Factory per oggetti Transaction. Non è esposta alla applicazione ma può essere estesa/implementata dallo sviluppatore.

## Oggetti persistent

Oggetti single-threaded che contengono lo stato persistente e la logica di business che possono essere normali JavaBean e devono essere associati esattamente a a un oggetto Session.

Modifiche fatte sugli oggetti persistenti sono automaticamente propagate sulle tabelle DB.

Appena si chiude la sessione, tali oggetti diventano "**detached**" e possono essere usati liberamente

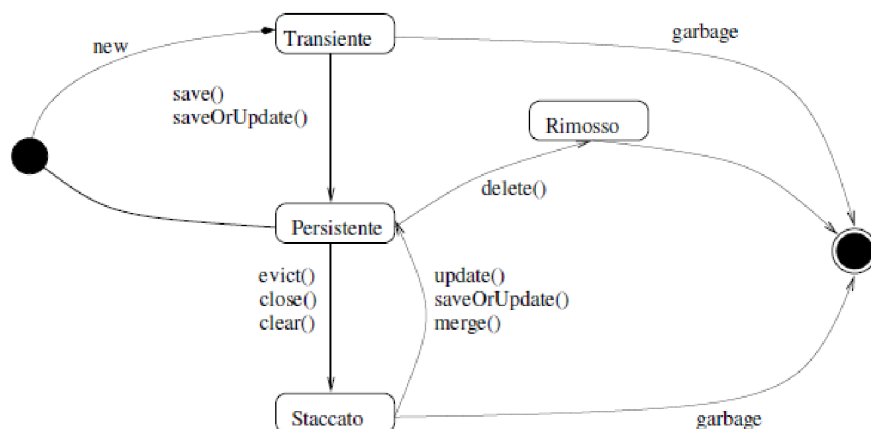
## Oggetti transient e detached

Istanze di classi persistenti che non sono correntemente associati a nessuna Session.

Modifiche fatte sugli questi oggetti NON si riflettono sul DB: occorrono operazioni di **persist** o **merge** per farli tornare "persistenti".

Gli oggetti transient non hanno identità di persistenza (valore associato alla primary key) nè di conseguenza righe corrispondenti nel DB mentre gli oggetti detached hanno identità di persistenza in quanto in passato sono stati associati a una riga del DB.

## Ciclo di vita degli oggetti persistenti



## Strategie di fetching

Una strategia di fetching determina come e quando i dati vengono effettivamente caricati dal DB per una applicazione che usa gli oggetti di persistenza associati. La strategia adottata ha ovviamente impatto sulle performance ottenibili e viene di solito dichiarata in un file di mapping. Modalità di fetching:

- FetchMode.DEFAULT – configurazione del mapping file
- FetchMode.JOIN – recupera i dati associati, anche collezioni, utilizzando un join all'interno della stessa select
- FetchMode.SELECT – effettua una seconda select separata per recuperare le entity o collection associate (lazy fetching: la seconda select viene eseguita solo quando l'applicazione accede veramente ai dati associati)

## Strategie di caching

Utilizzato per motivi di performance. L'idea è quella di rendere l'accesso al DB necessario solo quando si devono reperire dati che non sono già disponibili sulla cache (l'applicazione avrà bisogno di svuotare la cache se il DB viene aggiornato)

- First-level cache – associata con l'oggetto session (limite ad una singola transazione, principalmente al fine di ridurre il numero di query SQL generate all'interno di una transazione quando ad esempio un oggetto viene modificato diverse volte all'interno della medesima transazione)
- Second-level cache – associata con l'oggetto SessionFactory (oggetti persistenti disponibili per l'intera applicazione)

## Hibernate in azione

I principali elementi di un'applicazione che fa uso di Hibernate per la persistenza sono:

- classi di dominio realizzate in java
- base di dati (DB2)
- un file XML che definisce il mapping di ogni classe persistente (deve soddisfare la grammatica specificata all'interno di un apposito DTD chiamato `hibernate-mapping-3.0.dtd`)

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="User" table="UTENTE">                                <!-- Nome della
  classe persistente -->
    <id name="username" column="username"></id>                    <!-- Attributo chiave
  primaria-->
    <property name="nome" column="nome"></property>
    <property name="cognome" column="cognome"></property>
    <property name="email"></property>
    <property name="password"></property>
  </class>
</hibernate-mapping>
```

`<id>` può avere un elemento figlio `<generator>` che deve avere un attributo `class` che specifica una classe java che implementa l'interfaccia **org.hibernate.id.IdentifierGenerator**; Hibernate viene fornito con delle classi built-in per generare identificatori:

- assigned: significa che l'applicazione assegna l'identificatore all'istanza prima che sia chiamato il metodo
- save – questo è il comportamento di default se generator non compare
- native: usa i generatori di identificatori forniti dal DBMS sottostante; mantiene il mapping portabile

Possono però esserci più tipi Hibernate per uno stesso tipo Java: bisogna allora indicare nel file di mapping il tipo di mapping che Hibernate deve usare.

Quando invece una proprietà è una collezione omogenea di valori gestiti da una altra tabella, allora nel file di mapping si definisce un elemento apposito, quale Set, List (vedi documentazione).

- uno o più file di configurazione di Hibernate (deve soddisfare la grammatica specificata nel DTD chiamato `hibernate-configuration-3.0.dtd`)

```
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE hibernate-configuration
PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="hibernate.connection.driver_class">
com.ibm.db2.jcc.DB2Driver</property>
    <property name="hibernate.connection.url">
jdbc:db2://diva:deis.unibo.it:50000/tw_stud</property>
    <property name="connection.username">*****</property>
    <property name="connection.password">*****</property>

    <!-- SQL dialect -->
    <property name="dialect"> org.hibernate.dialect.DB2Dialect</property>
    <!-- JDBC connection pool (use C3P0) -->
    <property name="c3p0.min_size">5</property>
    <property name="c3p0.max_size">20</property>
    <property name="c3p0.timeout">300</property>
    <property name="c3p0.max_statement">50</property>
    <!-- Show and print nice SQL on stdout -->
    <property name="show_sql">true</property>
    <property name="format_sql">true</property>
    <!-- List of XML mapping files -->
    <mapping resource="User.hbm.xml" />
  </session-factory>
</hibernate-configuration>
```

- interfacce Hibernate per l'accesso alla base di dati

## Transazioni

La transazione è un'unità di elaborazione che gode delle proprietà: Atomicità, Consistenza, Isolamento, Durabilità (ACID).

**Atomicità**: una transazione è un'unità atomica (tutto o niente) di elaborazione. Fondamentale per garantire l'integrità dei dati. Se qualcosa va storto il sistema deve essere in grado di annullare tutti i cambiamenti (**abort** o **rollback**).

**Consistenza**: la transazione rispetta i vincoli di integrità quindi se lo stato iniziale è corretto anche lo stato finale lo è.

**Isolamento**: la transazione non risente delle altre transazioni concorrenti. Una transazione non espone i suoi stati intermedi

**Durabilità**: Gli effetti di una transazione andata in commit non vanno perduti, anche in presenza di guasti.

## Controllo di concorrenza

Due azioni sullo stesso oggetto si dicono essere in conflitto se almeno una delle due è una operazione di scrittura:

- conflitto Write-Read (W-R): T2 legge un dato precedentemente scritto da T1 (non ancora conclusa) -> può generare letture sporche

- conflitto Read-Write (R-W): T2 scrive un dato precedentemente letto da T1 (non ancora conclusa) -> perdita di aggiornamento, lettura inconsistente
- conflitto Write-Write (W-W): T2 scrive un dato precedentemente scritto da T1 (non ancora conclusa) -> effetti fantasma

I DBMS per evitare anomalie nelle transazioni concorrenti usano diverse tecniche; una di quelle più comuni è basata su **lock** (a livello di riga, tabella, database). Quando una risorsa è bloccata le transazioni che ne richiedono l'accesso vengono messe in coda.

## Livelli di isolamento

idealmente vorremmo avere sempre garantita la proprietà di isolamento delle transazioni ma questa ha dei costi che possono limitare le prestazioni del sistema. I DBMS offrono diversi livelli isolamento:

livello di isolamento	lettura sporca	lettura inconsistente	effetto fantasma
<b>READ UNCOMMITTED</b>	può verificarsi	può verificarsi	può verificarsi
<b>READ COMMITTED</b>	NO	può verificarsi	può verificarsi
<b>REPEATABLE READ</b>	NO	NO	può verificarsi
<b>SERIALIZABLE</b>	NO	NO	NO

## Transazioni in JDBC

```
connection.setAutoCommit(true) //ogni operazioni è una transazione
connection.setAutoCommit(false) //gestione delle transazioni da programma mediante
connection.commit() / connection.rollback()
```

**Isolamento:** L'isolamento si applica con il metodo `setTransactionIsolation(int)` passando come parametro costanti definite dall'interfaccia `connection`

```
connection.setTransactionIsolation(Connection.TRANSACTION_NONE); //le transazioni
non vengono supportate (è equivalente ad una chiamata di setAutoCommit(true))

//oppure

connection.setTransactionIsolation(Connection.TRANSACTION_READ_UNCOMMITTED);
connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
connection.setTransactionIsolation(Connection.TRANSACTION_REPEATABLE_READ);
connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
```

Attenzione: non tutti i DBMS supportano le transazioni e non tutti quelli che le supportano permettono l'impostazione di tutti i livelli di isolamento previsti da JDBC; è compito del programmatore che usa l'API JDBC verificare effettive capacità del DBMS attraverso il metodo

```
connection.getMetaData()
```

che restituisce un oggetto **DatabaseMetaData** che offre metodi per reperire informazioni sul DBMS:

```
supportsTransactions() // restituisce true se le transazioni sono supportate, false  
altrimenti (in questo caso l'isolamento è sempre TRANSACTION_NONE)  
supportsTransactionIsolationLevel(int) // permette di sapere se il sistema supporta  
il livello di isolamento passato come parametro secondo le costanti definite  
dall'interfaccia Connection
```

JDBC rappresenta una buona soluzione per la gestione di transazioni che manipolano un unico DB, ovvero che coinvolgono una unica Connection. In sistemi che manipolano dati appartenenti a diversi DB non è possibile garantire la proprietà di atomicità con JDBC.

### Java Transaction API (JTA)

Java Transaction API (JTA) fornisce un servizio per la gestione di transazioni distribuite per la piattaforma J2EE.

JTA permette l'interazione con lo sviluppatore mediante l'interfaccia `javax.transaction.UserTransaction` e i metodi `begin()`, `commit()` e `rollback()`

### Transazioni in Hibernate

Permettono di lavorare "on top" a JDBC/JTA e permette stretta integrazione con il contesto di persistenza (es. flush automatico di Session al commit).

All'apertura di un nuovo oggetto della classe Session viene creata una nuova connessione JDBC con la base di dati di default (Hibernate imposta l'autocommit a false dunque comincia a tutti gli effetti una transazioni JDBC).

La session di Hibernate è pigra cioè consuma la risorsa solo nel momento in cui è strettamente necessaria, dunque solo all'inizio di una transazioni viene resa disponibile una Connection dal pool di connessioni

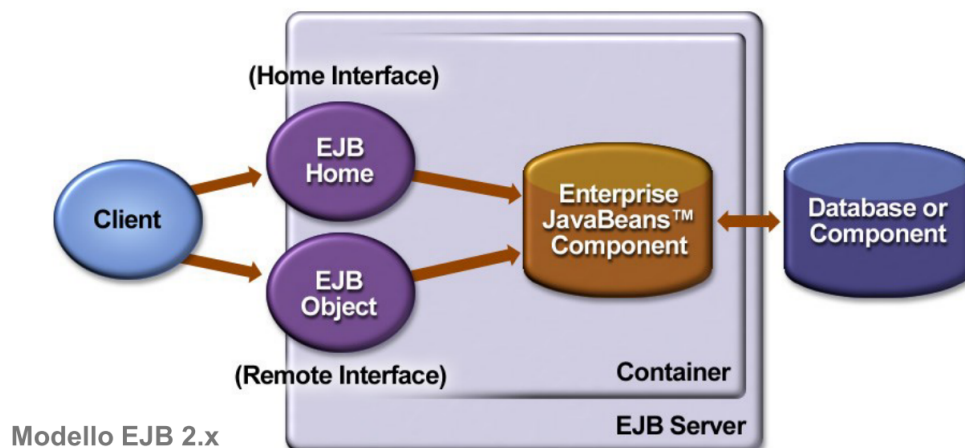
## Java Model 2, J2EE, EJB

Model 1 è un pattern semplice in cui il codice responsabile per presentazione contenuti è mescolato con logica di business (obsoleto nella pratica industriale). Model 2 è più complesso e articolato che separa chiaramente livello presentazione dei contenuti dalla logica utilizza per manipolare e processare contenuti stessi. Esso è associato con paradigma Model-View-Controller.

### EJB

Idea di base: container pesante attivo all'interno di un EJB Server (Application Server).

Cliente può interagire remotamente con componente EJB tramite interfacce ben definite passando SEMPRE attraverso container.



### Session Bean

Lavorano tipicamente per un singolo cliente, non sono persistenti, non rappresentano dati in un DB. Si usano per modellare oggetti di processo o di controllo specifici per un particolare cliente.

Stateless: esegue una richiesta e restituisce risultato senza salvare alcuna informazione di stato relativa al cliente

Stateful: può mantenere stato specifico per un cliente

## Entity Bean

Forniscono una vista ad oggetti dei dati mantenuti in un database, tempo di vita non connesso alla durata delle interazioni con i clienti (permangono nel sistema fino a che i dati esistono nel database). Nella maggior parte dei casi sono componenti sincronizzati con relativi database relazionali. Accesso condiviso per clienti differenti

## Message-Driven Bean (MDB)

Svolgono il ruolo di consumatori di messaggi asincroni. Non possono essere invocati direttamente dai clienti. Privi di stato.

## Servizi container-based

In un modello a container pesante vengono supportati:

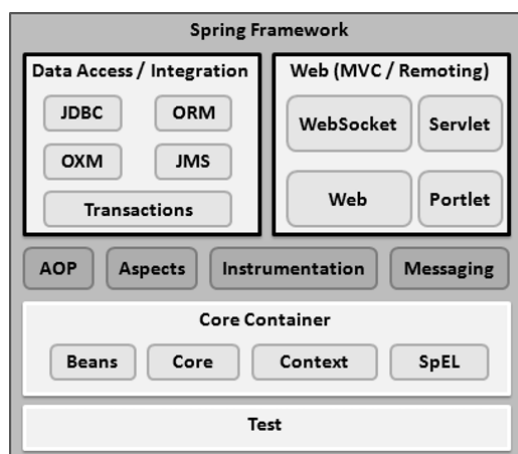
- Pooling e concorrenza
- Transazionalità
- Gestione delle connessioni a risorse
- Persistenza
- Messaggistica

# Spring

Spring è una tecnologia a container leggero java oriented. Funzionalità chiave:

- Inversion of control (IoC) e dependency injection
- Supporto alla persistenza
- Integrazione con Web tier
- Aspect Oriented Programming (AOP)

Per il mondo java è il primo container leggero, questo perché viene creato un core su cui si possono agganciare i soli moduli di interesse.



## Inversion of Control (IoC) o Dependency injection

Funzionalità che permette ai componenti di non dover cercare a runtime le risorse necessarie (su un server di nomi ad esempio). Le componenti dichiarano le risorse di cui hanno bisogno e il container si occupa di passarla al componente (inversion of control).

Bean factory è la factory per componenti che si occupa di ritrovare gli oggetti per nome e gestire le relazioni fra essi.

Per far in modo che Spring inietti all'interno del nostro componente una risorsa sarà necessario definire un costruttore con la dipendenza oppure i metodi get/set della risorsa. Ovviamente il tipo della dipendenza sarà un'interfaccia per non essere specifica per una singola classe.

```
public class SetterInjection{
    private Dependency dep;
    public void setMyDependency(Dependency dep){
        this.dep = dep;
    }
}
```

Per utilizzare il componente definito, il cliente dovrà sempre passare attraverso la Bean Factory. Solitamente la Bean Factory viene creata dall'applicazione tipicamente nella forma di XmlBeanFactory leggendo un file di configurazione XML configurato:

```
XmlBeanFactory f = new XmlBeanFactory(new FileSystemResource("bean.xml"));
SomeBeanInterface b = (SomeBeanInterface) f.getBean("nameOfTheBean");
```

Benefici derivanti dall'adozione di Dependency injection sono:

- flessibilità : non ci si dovrà concentrare su come reperire le informazioni ma implementare solo la logica di business.
- possibilità e facilità di testing: senza bisogno di cambiare il sorgente per cambiare la locazione dei dati
- manutenibilità: possibilità di riutilizzo in diversi ambienti applicativi

## Persistenza

Livello di astrazione per gestione delle transazioni con DB (integrato con Hibernate e DAO attraverso delle API).

## Integrazione con Web tier

Si utilizza il framework MVC per applicazioni web con supporto per diverse tecnologie per la generazione di viste.

## Supporto a Aspect Oriented Programming

Tecnica che permette di separare in moduli diversi i vari aspetti dell'applicazione in modo da aumentarne la modularità.

## Supporto a MVC in Spring

Spring mette a disposizione una servlet DispatcherServlet già pronta a cui il cliente può collegarsi. Attraverso dei file di configurazione essa sarà in grado di delegare le richieste al controller corretto. Il controller, di cui sono presenti già dei modelli all'interno di Spring, restituisce il model al dispatcher. Il dispatcher decide la vista associata e passa il modello ad essa. Quest'ultima restituisce il controllo al dispatcher che si incarica di produrre la risposta.

